# Graph Algorithms

## Graphs

- A graph $G = (V, E)$
  - $V$ = set of vertices
  - $E$ = set of edges = subset of $V \times V$
  - Thus $|E| = O(|V|^2)$

## Graph Variations

- Variations:
  - A *connected graph* has a path from every vertex to every other
  - In an *undirected graph:*
    - Edge (u,v) = edge (v,u)
  - In a *directed* graph:
    - Edge (u,v) goes from vertex u to vertex v, notated $u \rightarrow v$

## Graph Variations

- More variations:
  - A *weighted graph* associates weights with either the edges or the vertices
    - E.g., a road map: edges might be weighted w/ distance

## Graphs

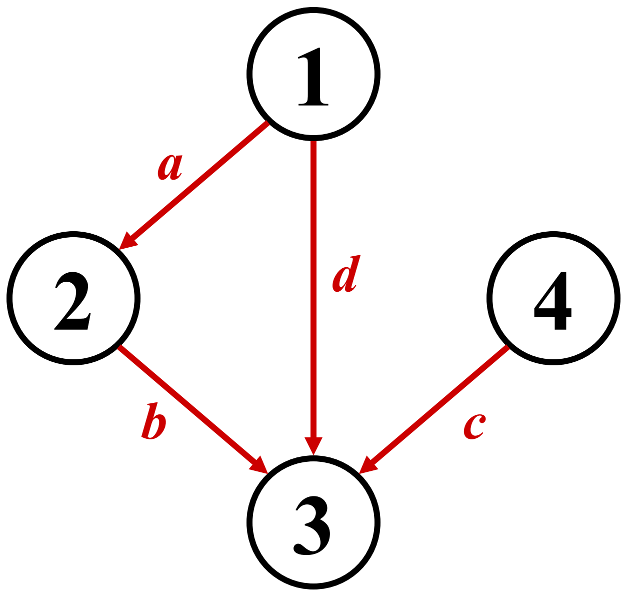
- We will typically express running times in terms of $|E|$ and $|V|$
  - If $|E| \approx |V|^2$ the graph is *dense*
  - If $|E| \approx |V|$ the graph is *sparse*
- If you know you are dealing with dense or sparse graphs, different data structures may make sense

## Representing Graphs

- Assume V = $\{1, 2, \ldots, n\}$
- An *adjacency matrix* represents the graph as a $n$ x $n$ matrix A:
  - $A[i, j] = 1$ if edge $(i, j) \in E$ (or weight of edge)
    $= 0$ if edge $(i, j) \notin E$

## Graphs: Adjacency Matrix

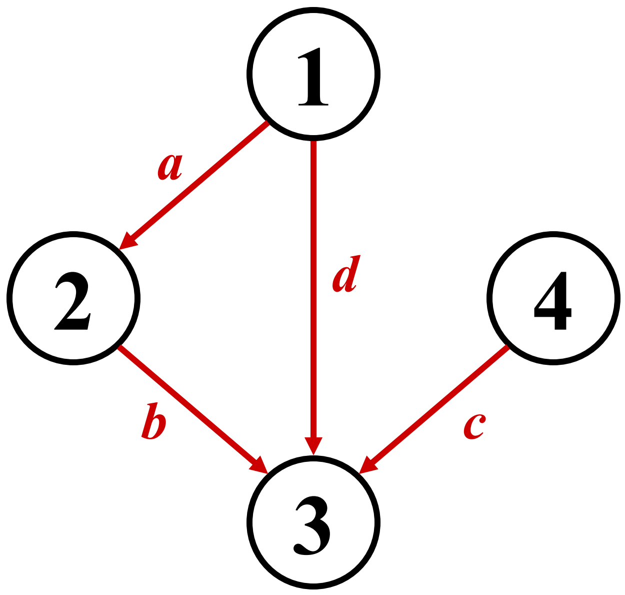
- Example:



| A | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | | | | |
| 2 | | | | |
| 3 | | | ?? | |
| 4 | | | | |

# Graphs: Adjacency Matrix

- Example:



| A | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 0 |
| 2 | 0 | 0 | 1 | 0 |
| 3 | 0 | 0 | 0 | 0 |
| 4 | 0 | 0 | 1 | 0 |

# Graphs: Adjacency Matrix

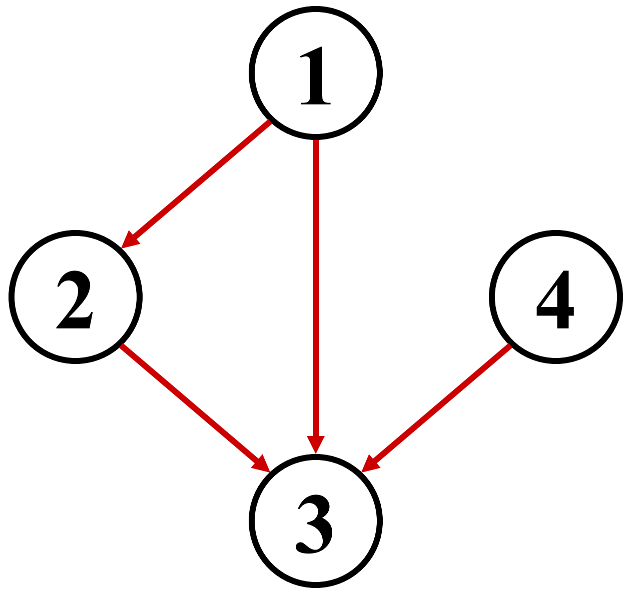
- *How much storage does the adjacency matrix require?*
- A: O($V^2$)
- *What is the minimum amount of storage needed by an adjacency matrix representation of an undirected graph with 4 vertices?*
- A: 6 bits
  - Undirected graph → matrix is symmetric
  - No self-loops → don't need diagonal

# Graphs: Adjacency Matrix

- The adjacency matrix is a dense representation
  - Usually too much storage for large graphs
  - But can be very efficient for small graphs
- Most large interesting graphs are sparse
  - For this reason the *adjacency list* is often a more appropriate respresentation

# Graphs: Adjacency List

- Adjacency list: for each vertex $v \in V$, store a list of vertices adjacent to $v$
- Example:
  - Adj[1] = {2,3}
  - Adj[2] = {3}
  - Adj[3] = {}
  - Adj[4] = {3}
- Variation: can also keep a list of edges coming *into* vertex



## Graphs: Adjacency List

- How much storage is required?
  - The *degree* of a vertex $v$ = # incident edges
    - Directed graphs have in-degree, out-degree
  - For directed graphs, # of items in adjacency lists is

    $$\Sigma \text{ out-degree}(v) = |E|$$

    takes $\Theta(V + E)$ storage
  - For undirected graphs, # items in adj lists is

    $$\Sigma \text{ degree}(v) = 2\,|E|$$

    also $\Theta(V + E)$ storage
- So: Adjacency lists take $O(V+E)$ storage

## Graph Searching

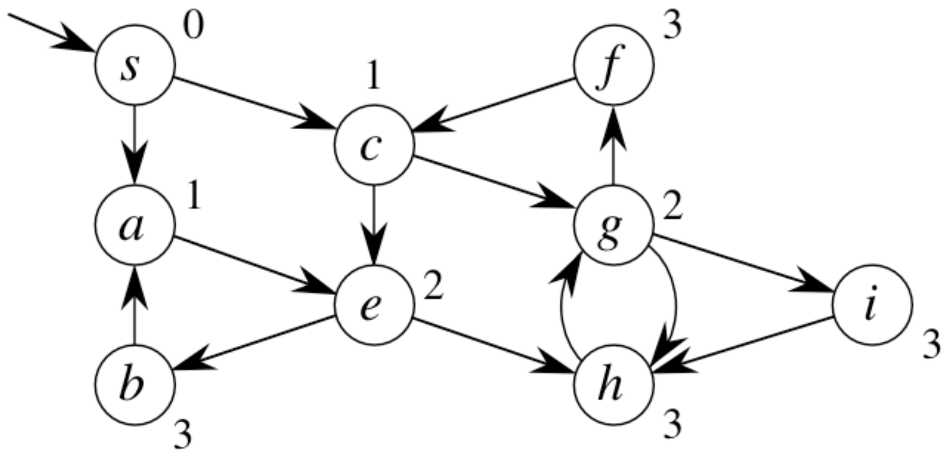
- Given: a graph G = (V, E), directed or undirected
- Goal: methodically explore every vertex and every edge
- Ultimately: build a tree on the graph
  - Pick a vertex as the root
  - Choose certain edges to produce a tree
  - Note: might also build a *forest* if graph is not connected

# Traverse of Graph

- Given graph $G = (V, E)$
- Given a source node $s \in V$
  - Visit all nodes in $V$ reachable from $s$
- Need an efficient strategy to achieve that
  - BFS: Breadth-first search

# Intuition

- Starting from source node *s*,
  - Spread a wavefront to visit other nodes
- Example
  - Imagine given a tree
  - What if a graph?



## Intuition cont.

- *$\delta(u, v)$:*
  - Distance (smallest # edges) from node $u$ to $v$ in $G$
- Goal:
  - Start from source $s$, first visit those nodes of distance $1$ from $s$, then $2$, and so on.
- *More formally: BFS*
  - *Input: Graph $G=(V, E)$ and source node $s \in V$*
  - *Output: $\delta(s, u)$ for every $u \in V$*
    - *$\delta(s,u) = \propto$ if $u$ is unreachable from $s$*
- *Assume adjacency list representation for $G$*

## Breadth-First Search

- "Explore" a graph, turning it into a tree
  - One vertex at a time
  - Expand frontier of explored vertices across the *breadth* of the frontier
- Builds a tree over the graph
  - Pick a *source vertex* to be the root
  - Find ("discover") its children, then their children, etc.

## Breadth-First Search

- Will associate vertex “colors” to guide the algorithm
  - White vertices have not been discovered
    - All vertices start out white
  - Grey vertices are discovered but not fully explored
    - They may be adjacent to white vertices
  - Black vertices are discovered and fully explored
    - They are adjacent only to black and gray vertices
- Explore vertices by scanning adjacency list of grey vertices

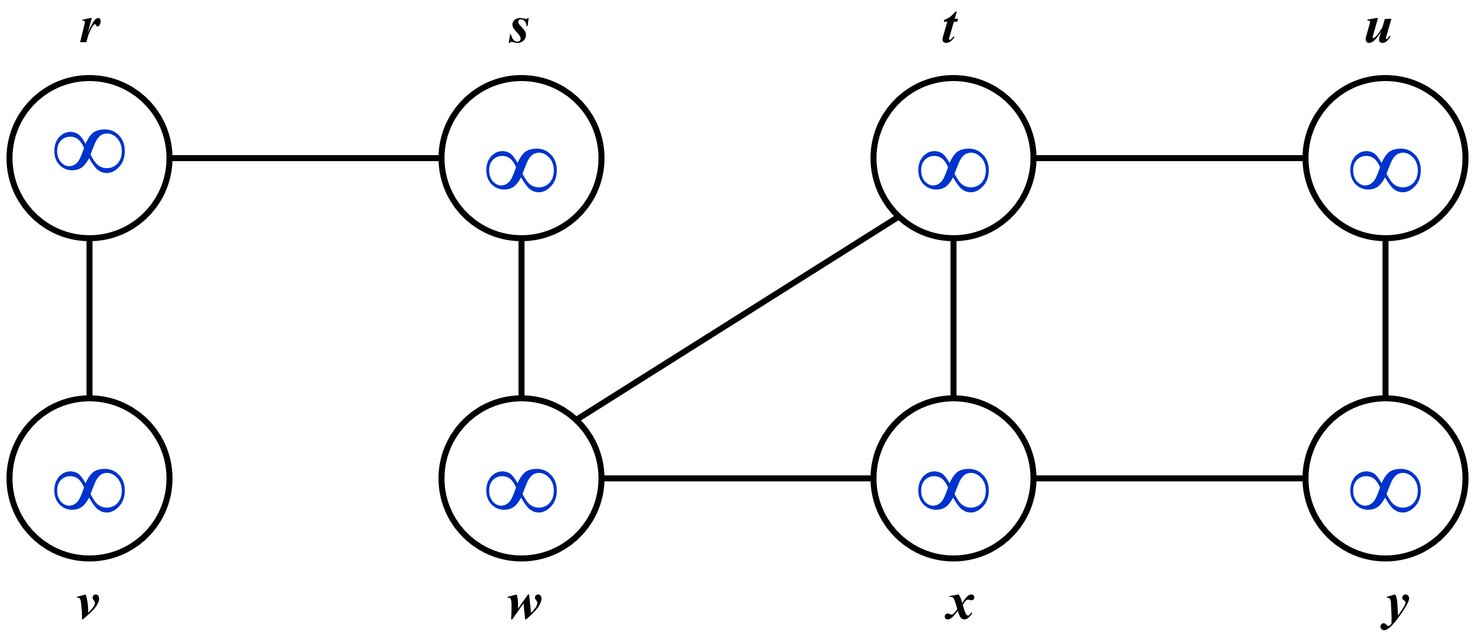
# Breadth-First Search

```
BFS(G, s) {
    initialize vertices;
    Q = {s};        // Q is a queue ; initialize to s
    while (Q not empty) {
        u = RemoveTop(Q);
        for each v ∈ u->adj {
            if (v->color == WHITE)
                v->color = GREY;
                v->d = u->d + 1;
                v->p = u;
                Enqueue(Q, v);
        }
        u->color = BLACK;
    }
}
```

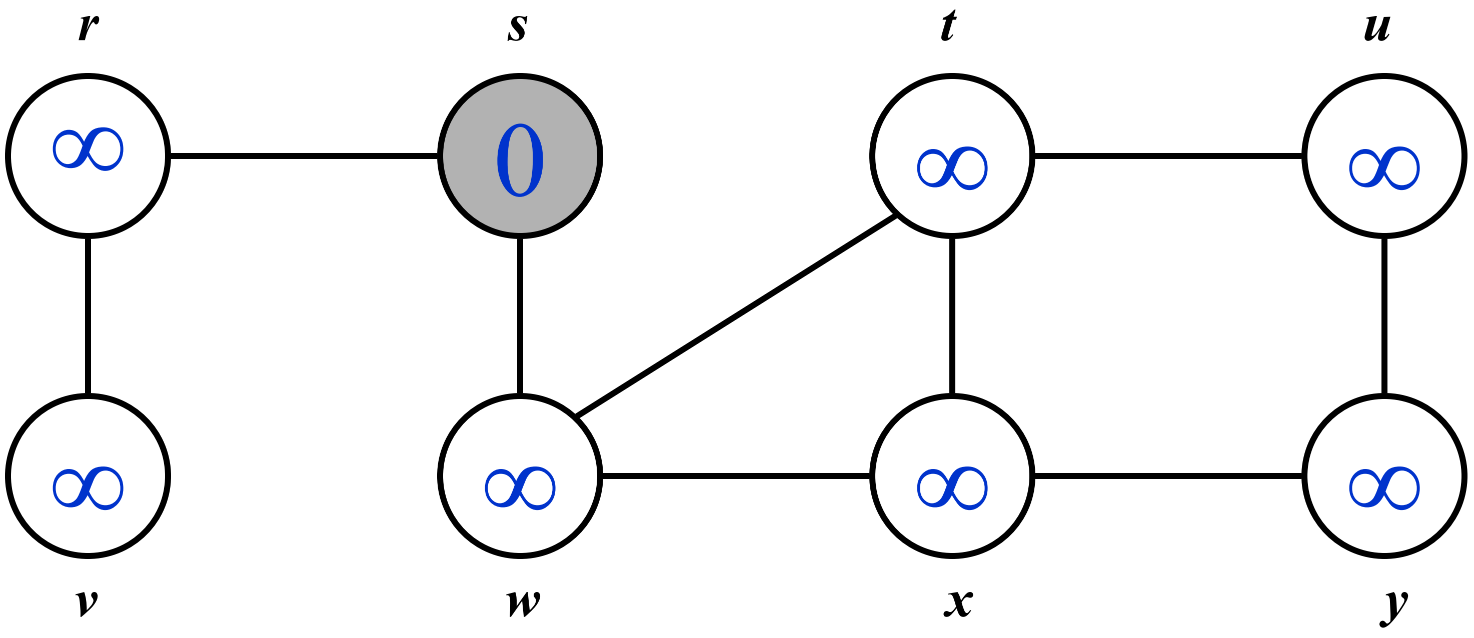
*What does* **`v->d`** *represent?*

*What does* **`v->p`** *represent?*

# Breadth-First Search: Example

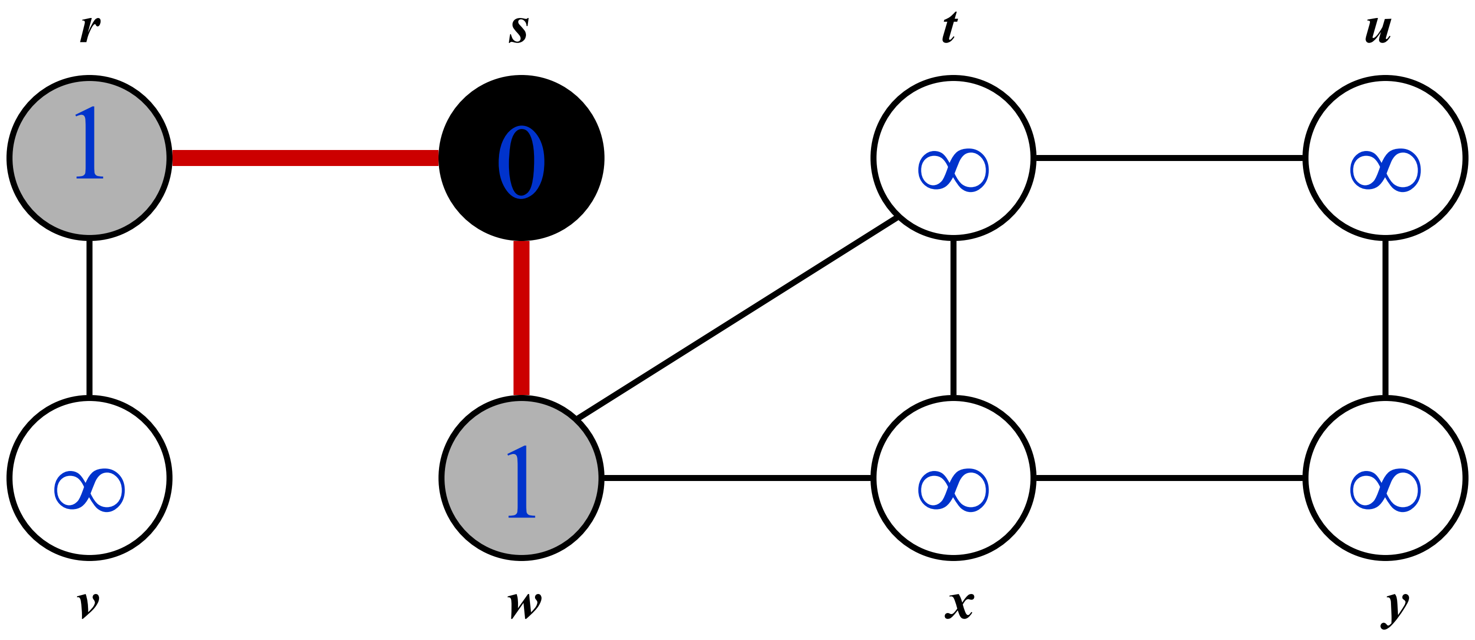


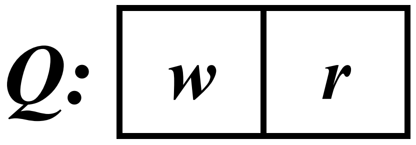
# Breadth-First Search: Example



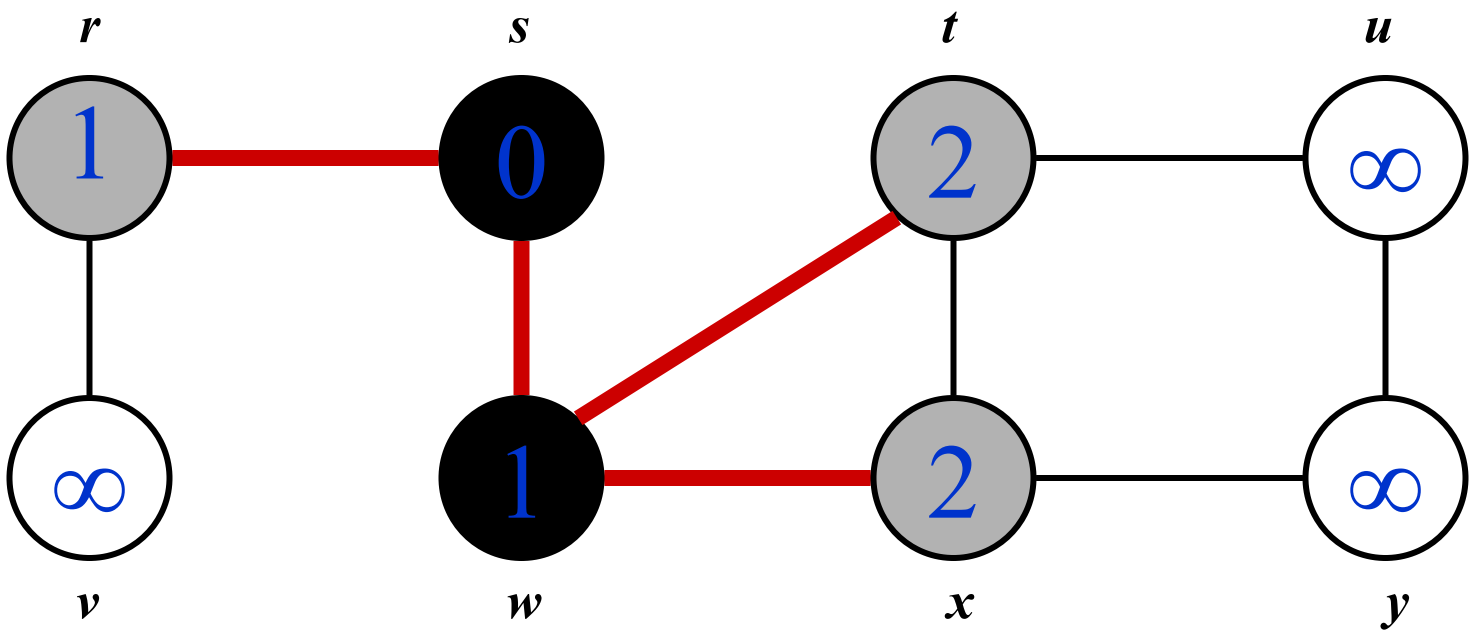
***Q:*** $\boxed{s}$

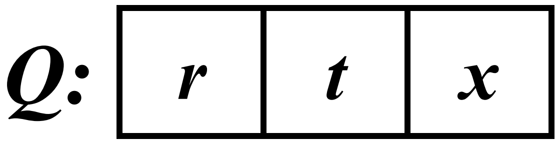
# Breadth-First Search: Example



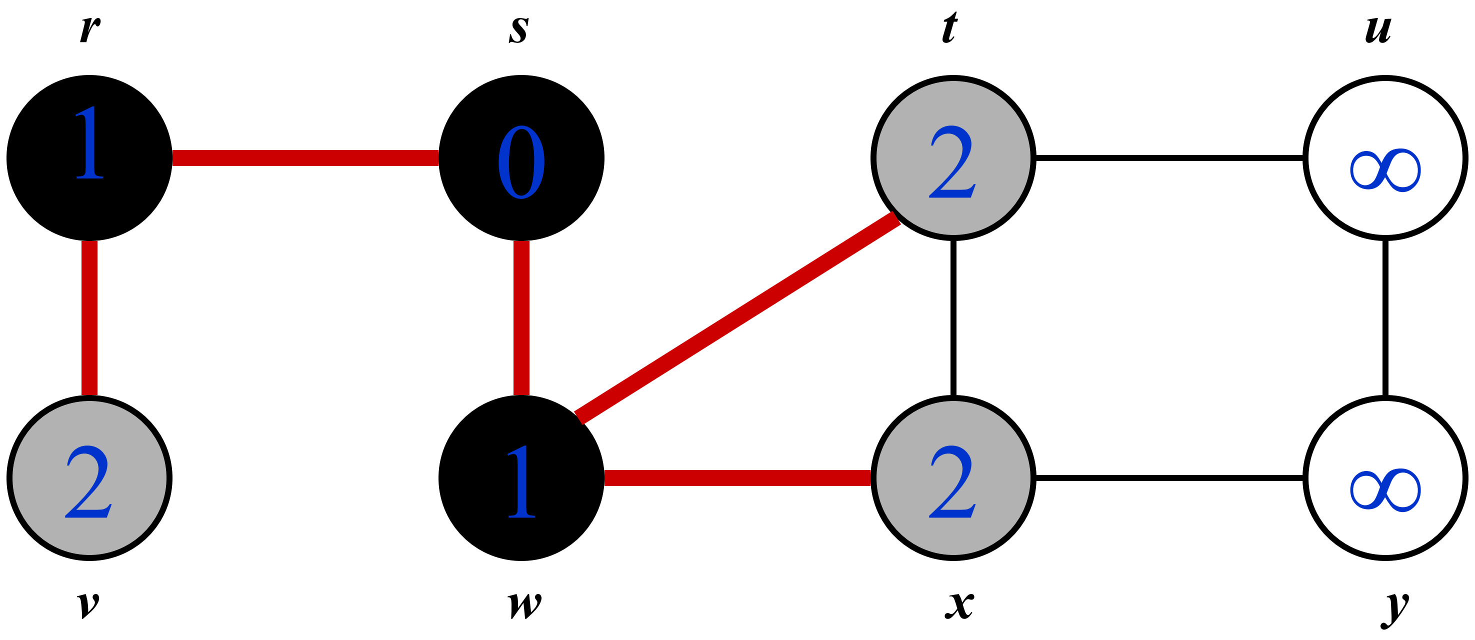


# Breadth-First Search: Example



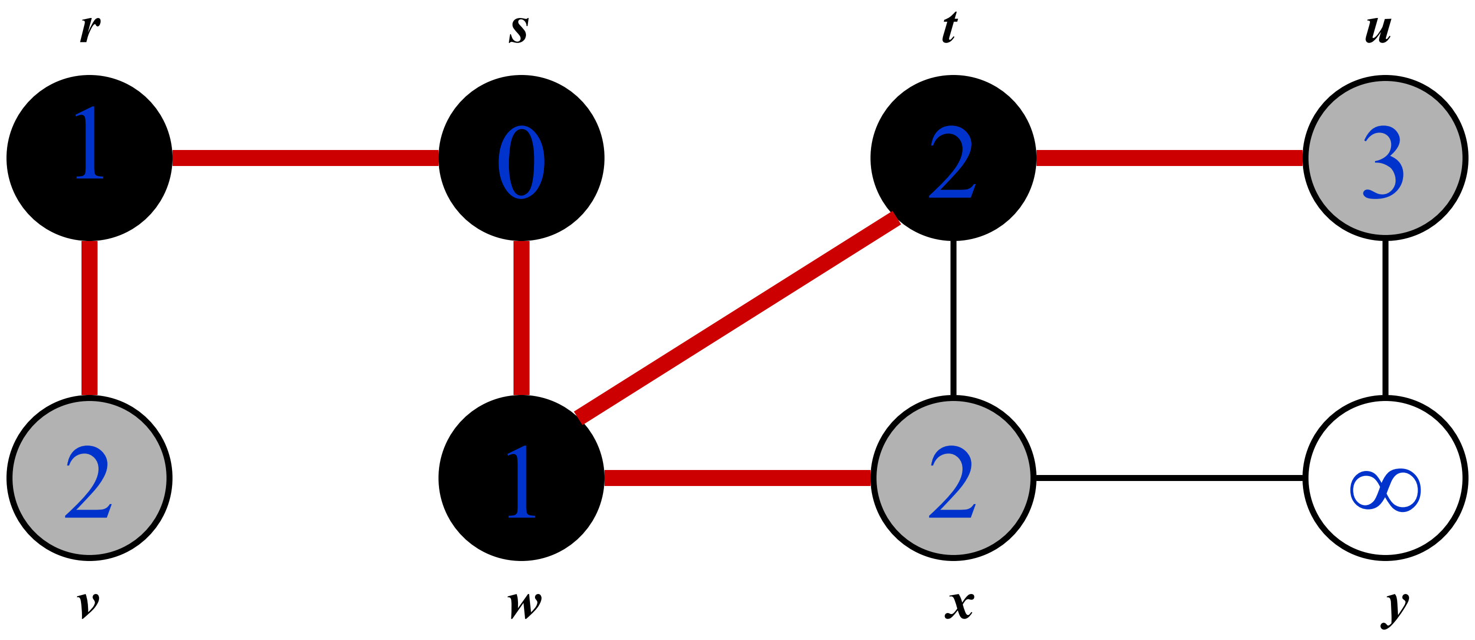


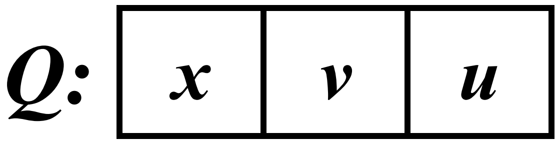
# Breadth-First Search: Example



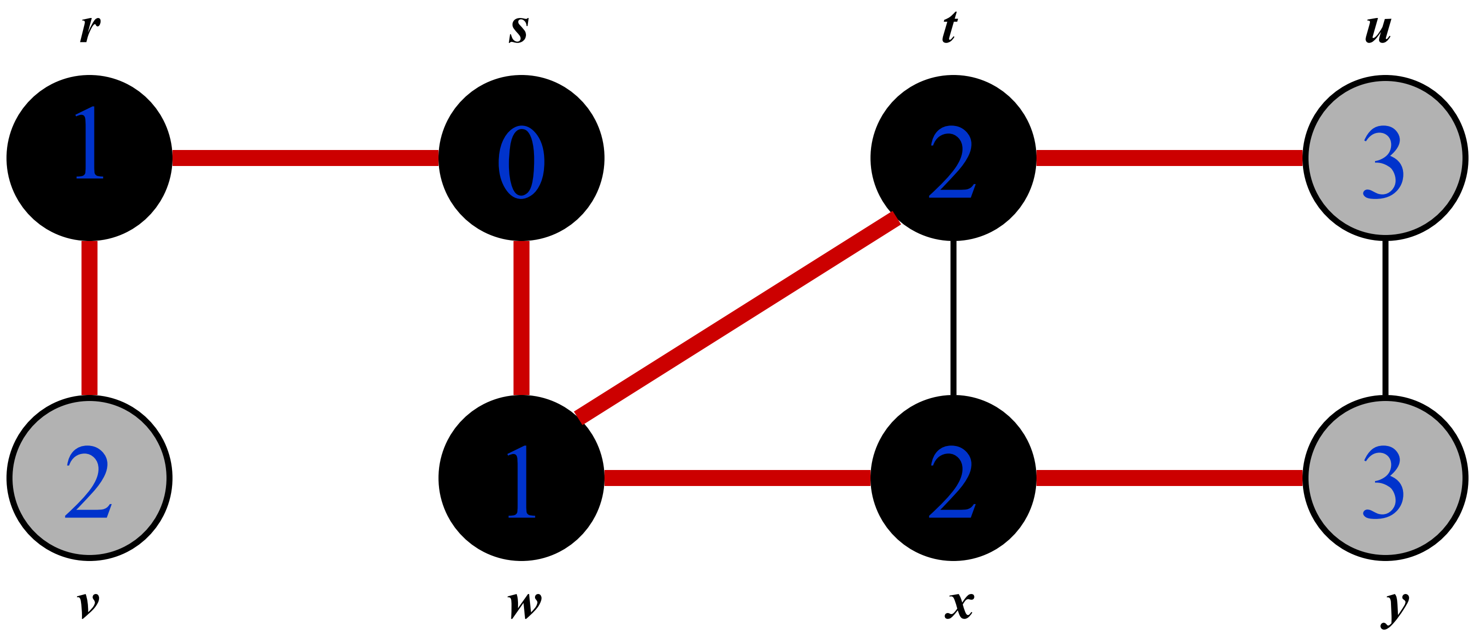
| $Q$: | $t$ | $x$ | $v$ |
|---|---|---|---|

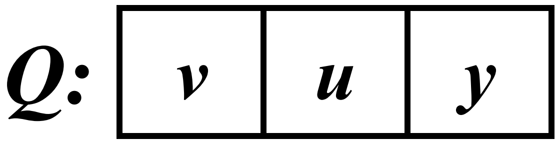
# Breadth-First Search: Example



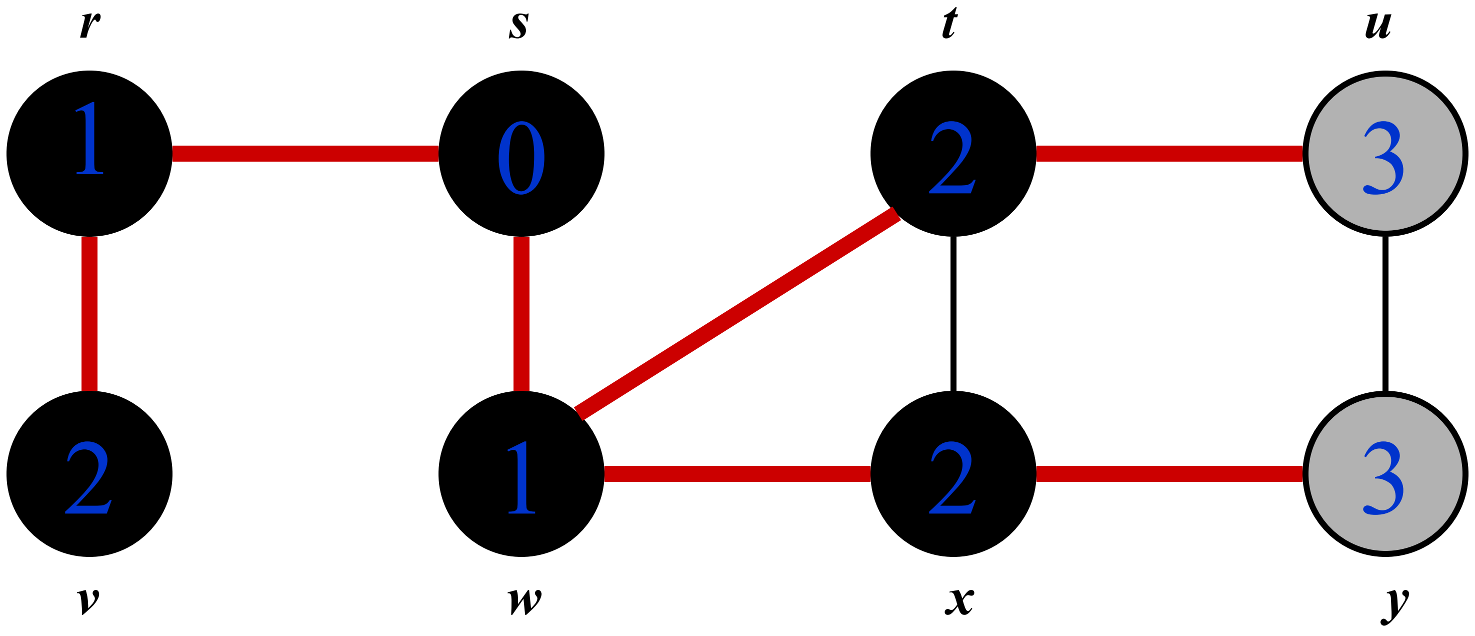


# Breadth-First Search: Example





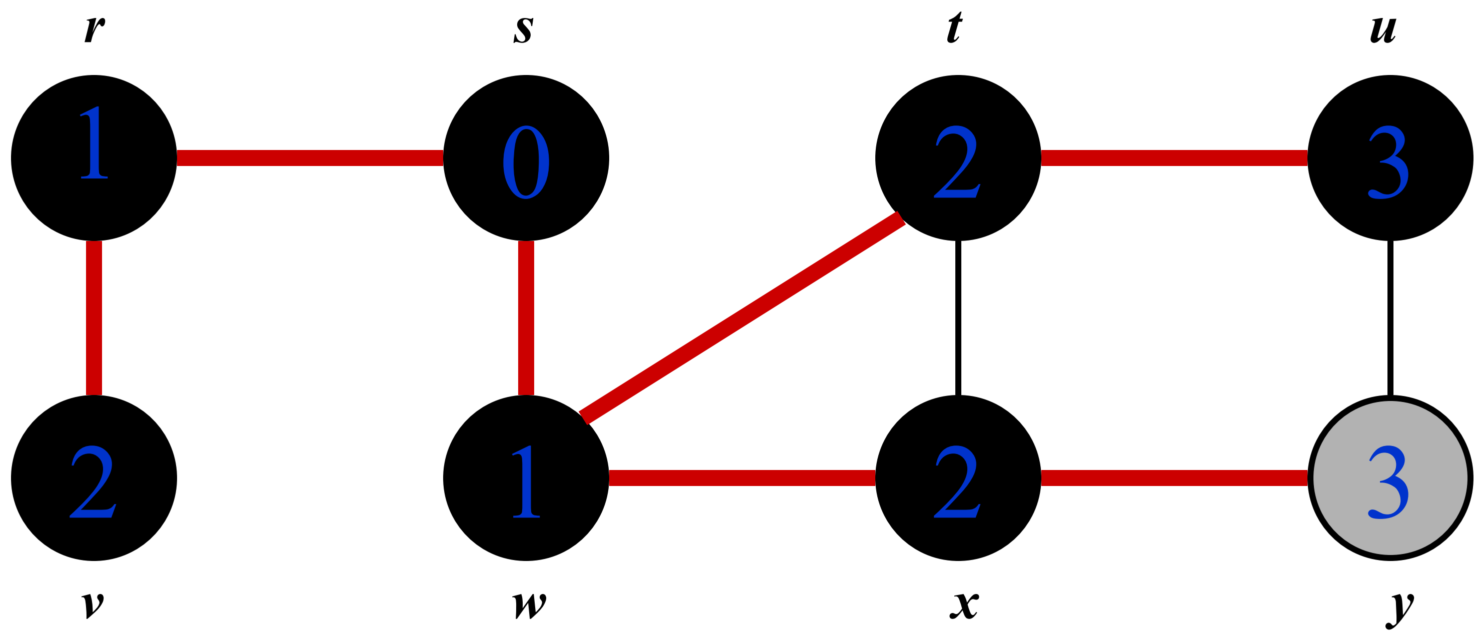
# Breadth-First Search: Example



*Q:*

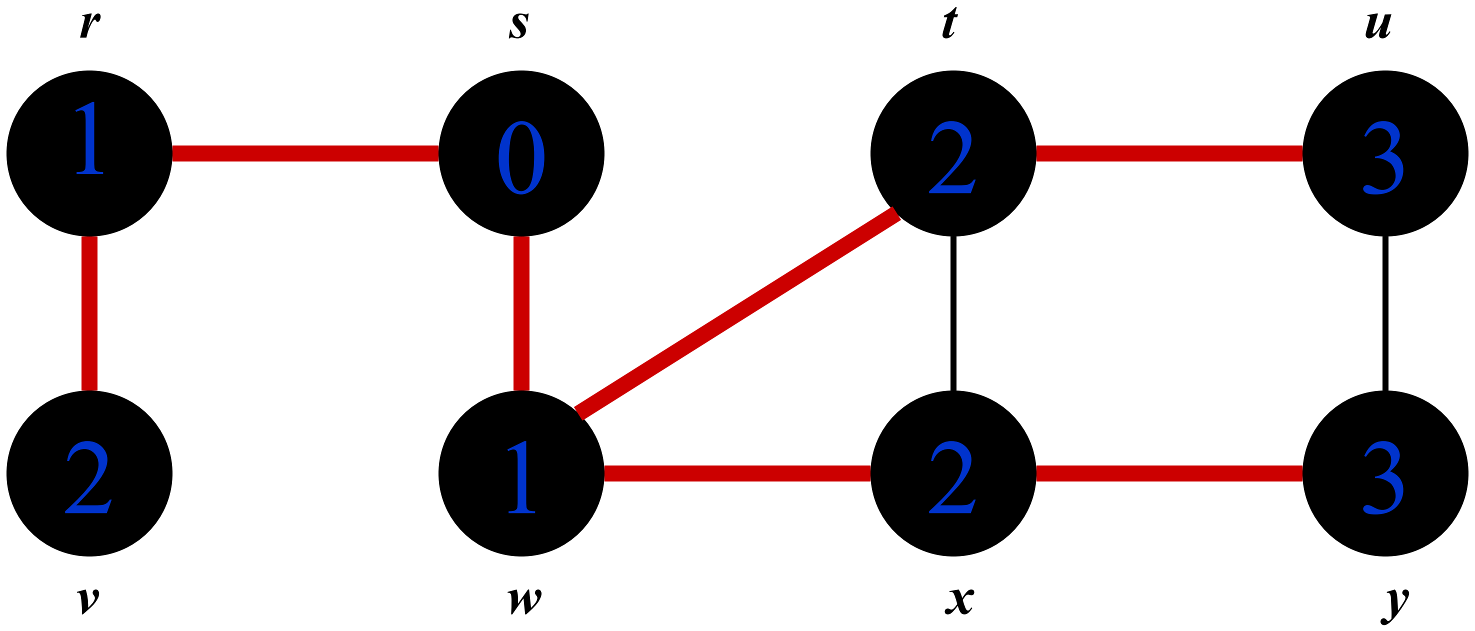
| | |
|---|---|
| *u* | *y* |

# Breadth-First Search: Example



*Q*: $\boxed{y}$

# Breadth-First Search: Example



$Q$: ∅

# BFS: The Code Again

```
BFS(G, s) {
    initialize vertices;
    Q = {s};
    while (Q not empty) {
        u = RemoveTop(Q);
        for each v ∈ u->adj {
            if (v->color == WHITE)
                v->color = GREY;
                v->d = u->d + 1;
                v->p = u;
                Enqueue(Q, v);
        }
        u->color = BLACK;
    }
}
```

*Touch every vertex: O(V)*

*u = every vertex, but only once*

*So v = every vertex that appears in some other vert's adjacency list*

*What will be the running time?*

**Total running time: O(V+E)**

# Breadth-First Search: Properties

- BFS calculates the *shortest-path distance* to the source node
  - Shortest-path distance $\delta(s,v)$ = minimum number of edges from s to v, or $\infty$ if v not reachable from s
- BFS builds *breadth-first tree*, in which paths to root represent shortest paths in G
  - Thus can use BFS to calculate shortest path from one vertex to another in $O(V+E)$ time

## Depth-First Search

- *Depth-first search* is another strategy for exploring a graph
  - Explore "deeper" in the graph whenever possible
  - Edges are explored out of the most recently discovered vertex $v$ that still has unexplored edges
  - When all of $v$'s edges have been explored, backtrack to the vertex from which $v$ was discovered

## Depth-First Search

- Vertices initially colored white
- Then colored gray when discovered
- Then black when finished

# Depth-First Search: The Code

```
DFS(G)
{
    for each vertex u ∈ G->V
    {
        u->color = WHITE;
    }
    time = 0;
    for each vertex u ∈ G->V
    {
        if (u->color == WHITE)
            DFS_Visit(u);
    }
}
```

```
DFS_Visit(u)
{
    u->color = GREY;
    time = time+1;
    u->d = time;
    for each v ∈ u->Adj[]
    {
        if (v->color == WHITE)
            DFS_Visit(v);
    }
    u->color = BLACK;
    time = time+1;
    u->f = time;
}
```

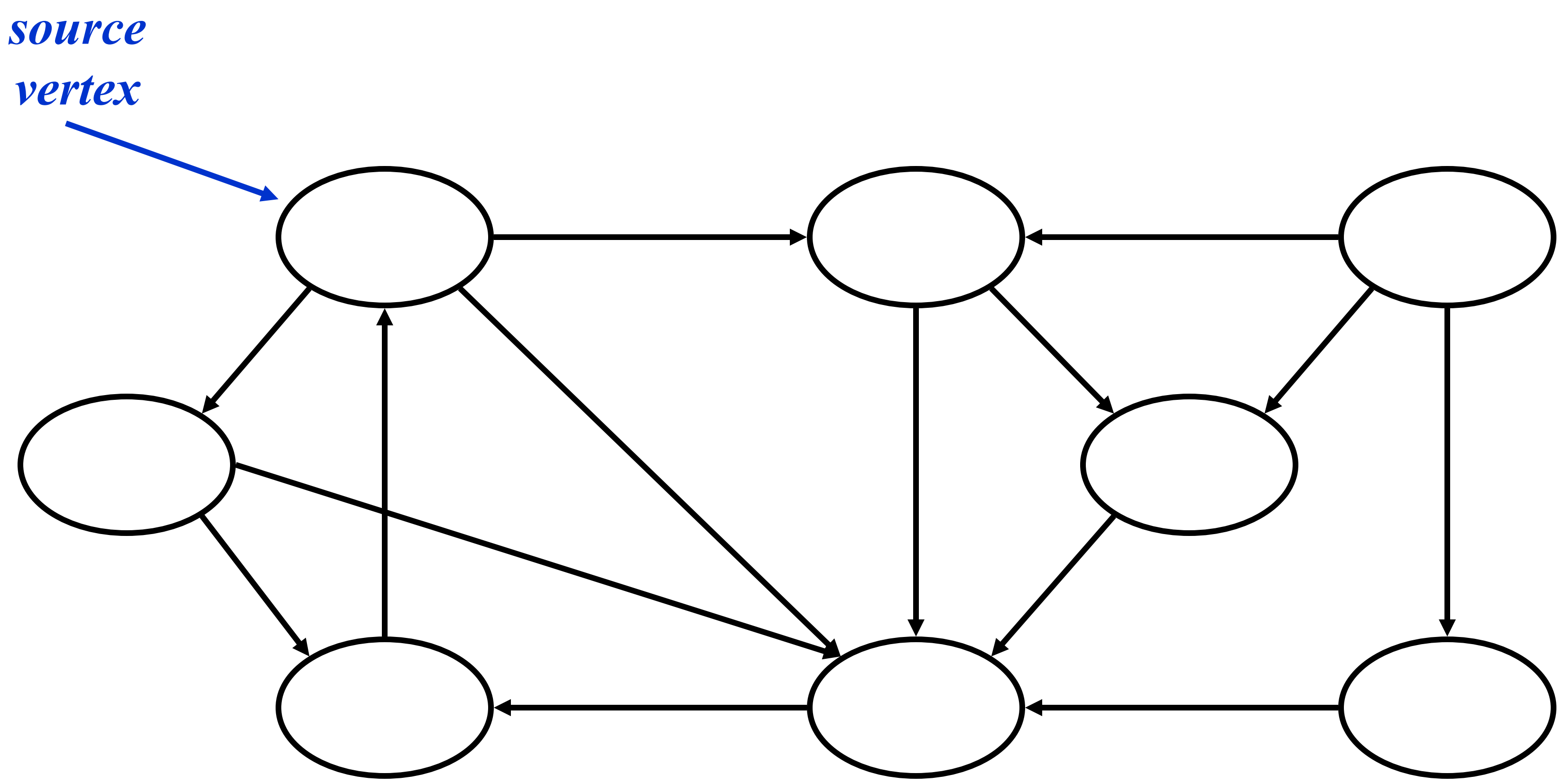
# Depth-First Search: Running Time

- *The loops on lines 1-3 and lines 5-7 of DFS take time* ***Θ(V),*** *exclusive of the time to execute the calls to DFS-VISIT.*

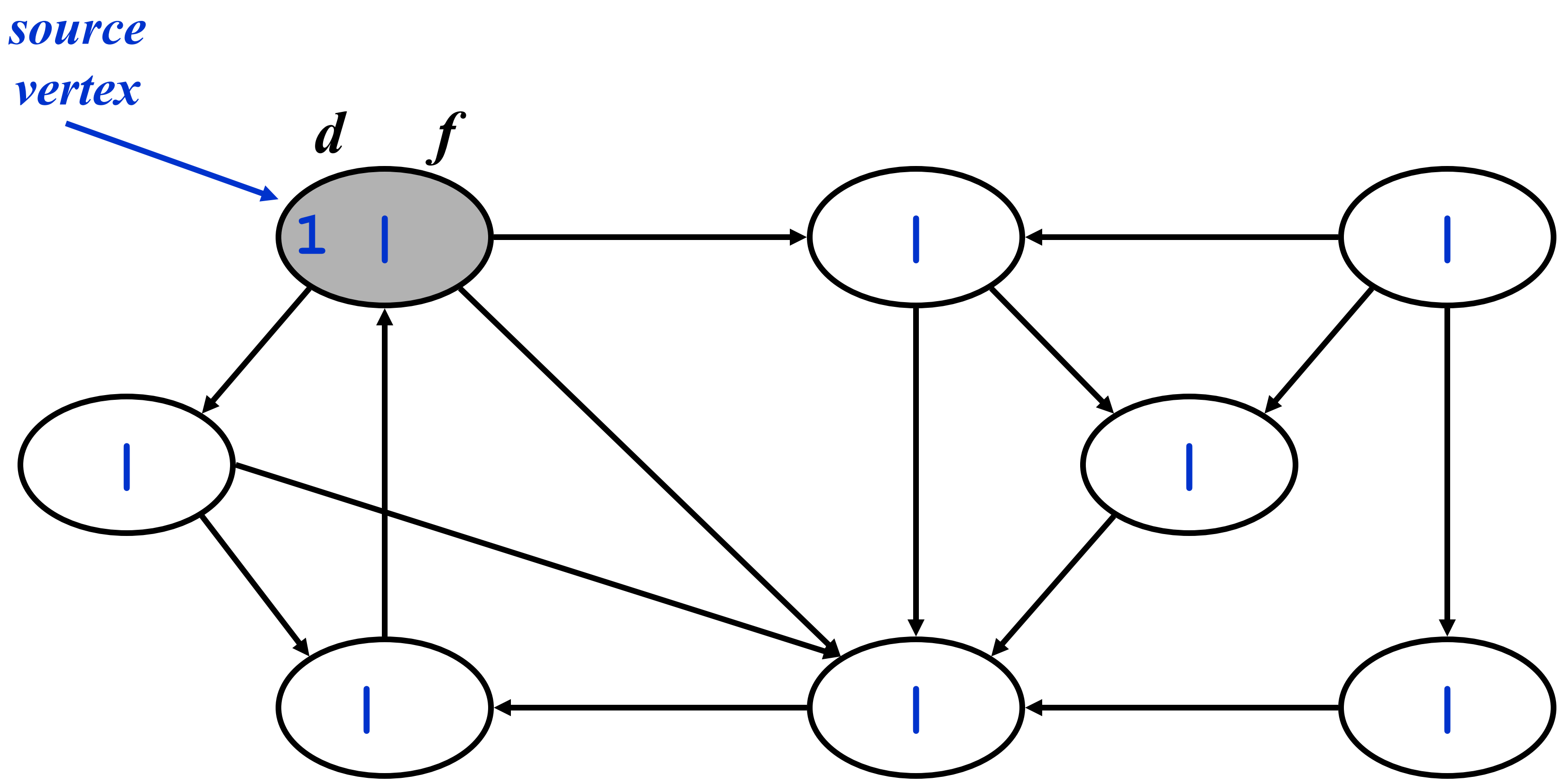
- *The procedure DFS-VISIT is called exactly once for each vertex v* □ *V , since DFS-VISIT is invoked only on white vertices and the first thing it does is paint the vertex gray.*

- *During an execution of DFS-VISIT(v), the loop on lines 4-7 is executed |Adj[v]| times. Since the total cost of executing lines 4-7 of DFS-VISIT is* ***Θ(E).*** *The running time of DFS is therefore* ***Θ(V + E).***

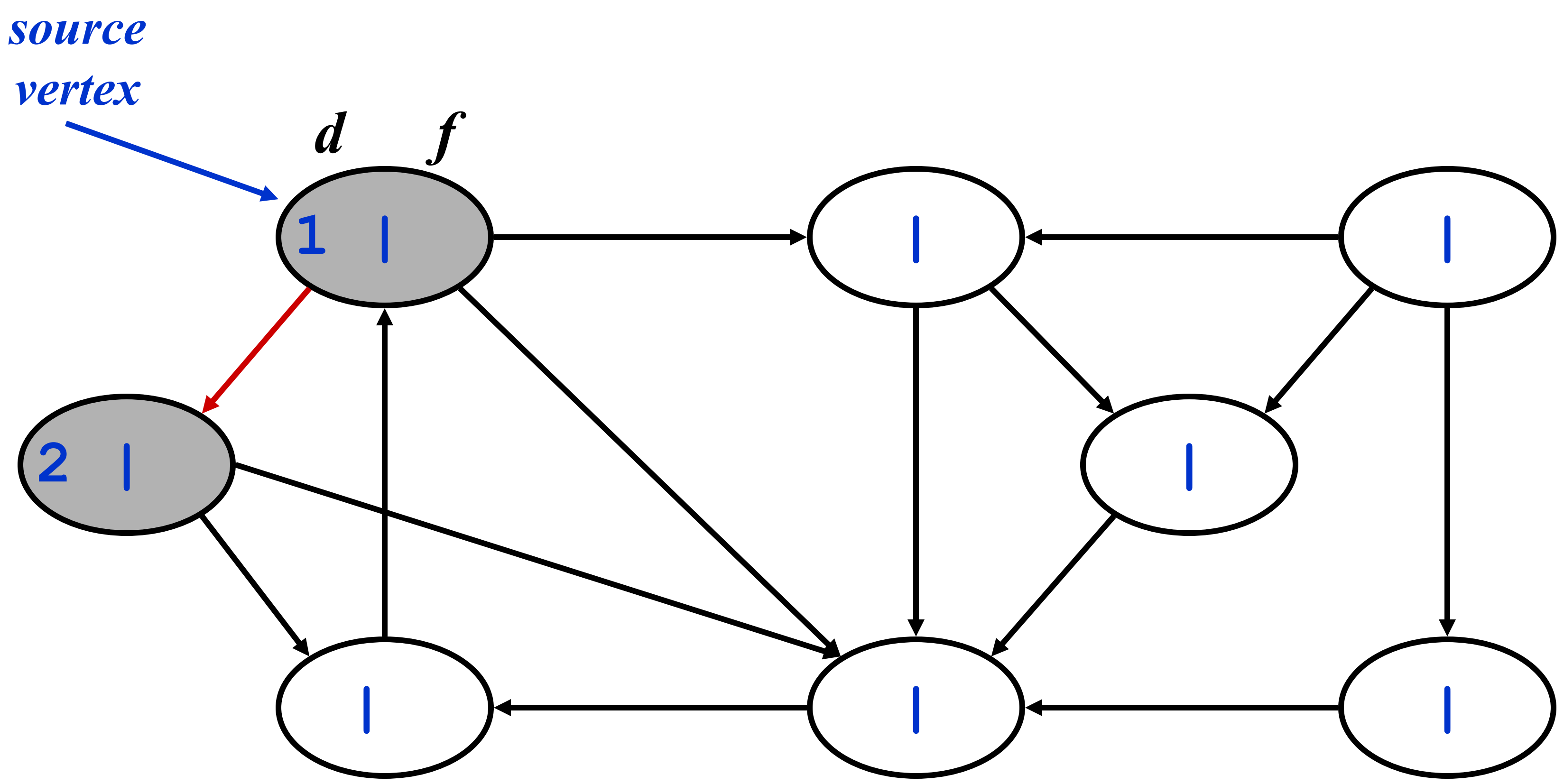
# DFS Example



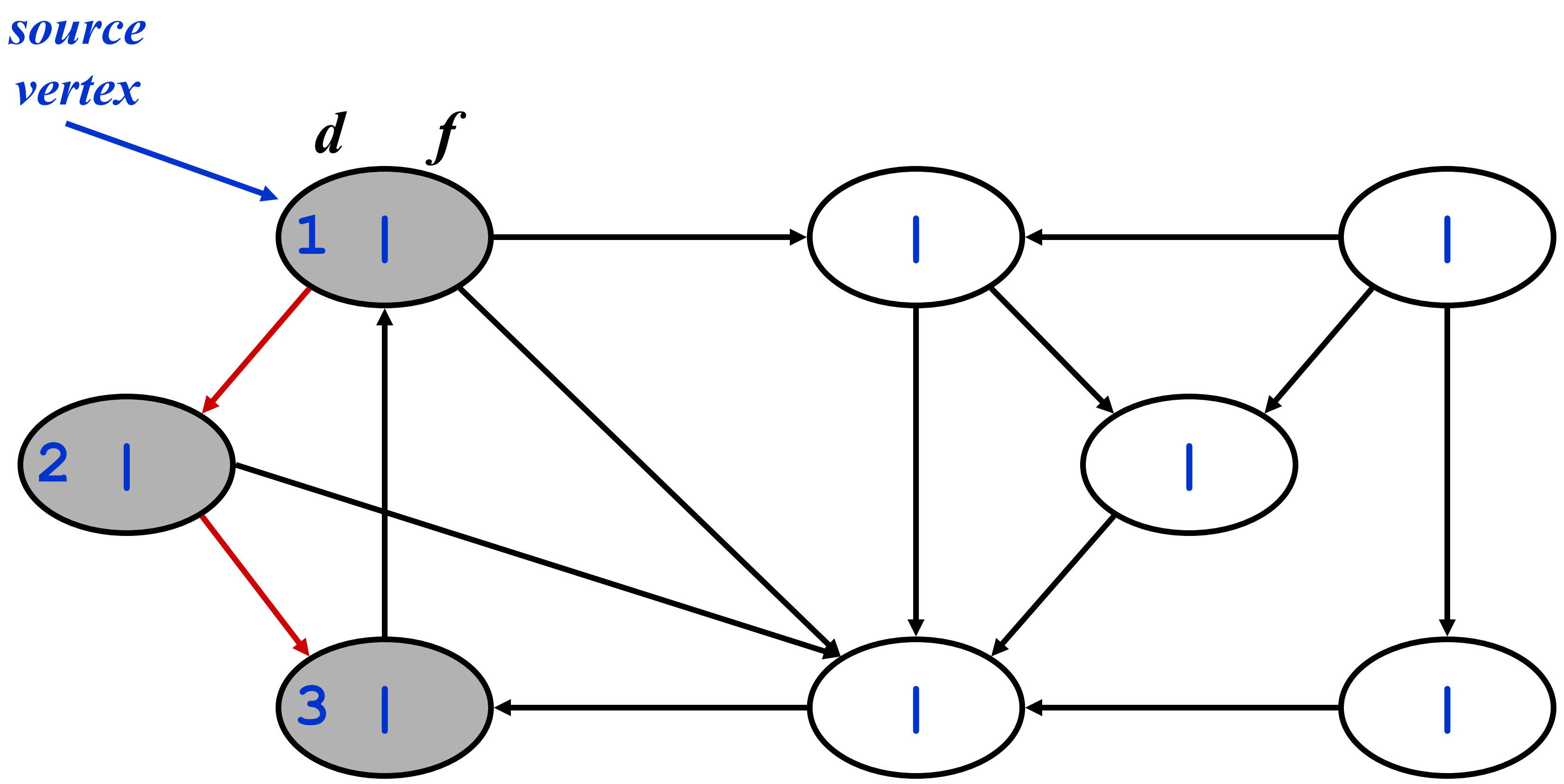
# DFS Example



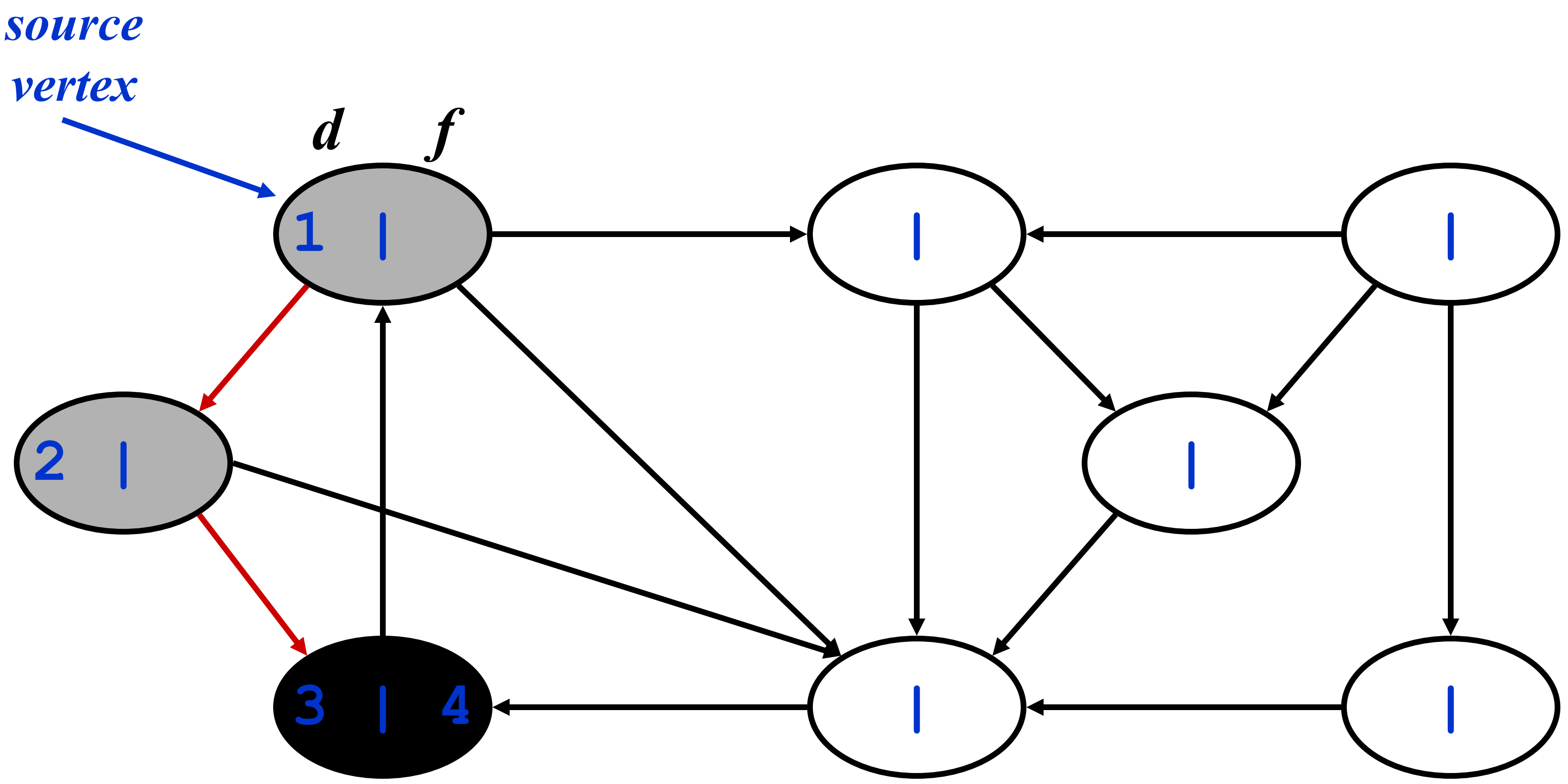
# DFS Example



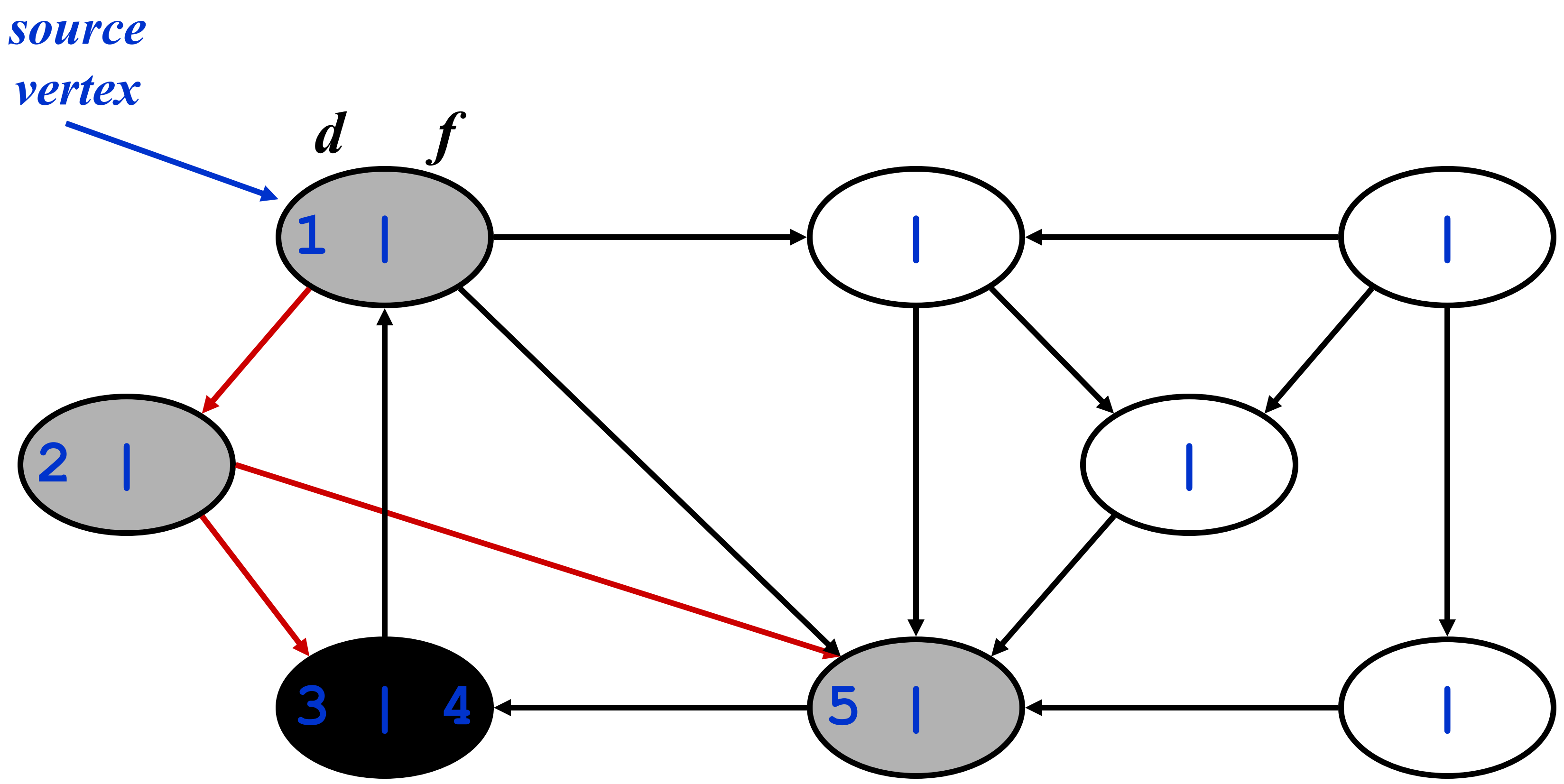
# DFS Example



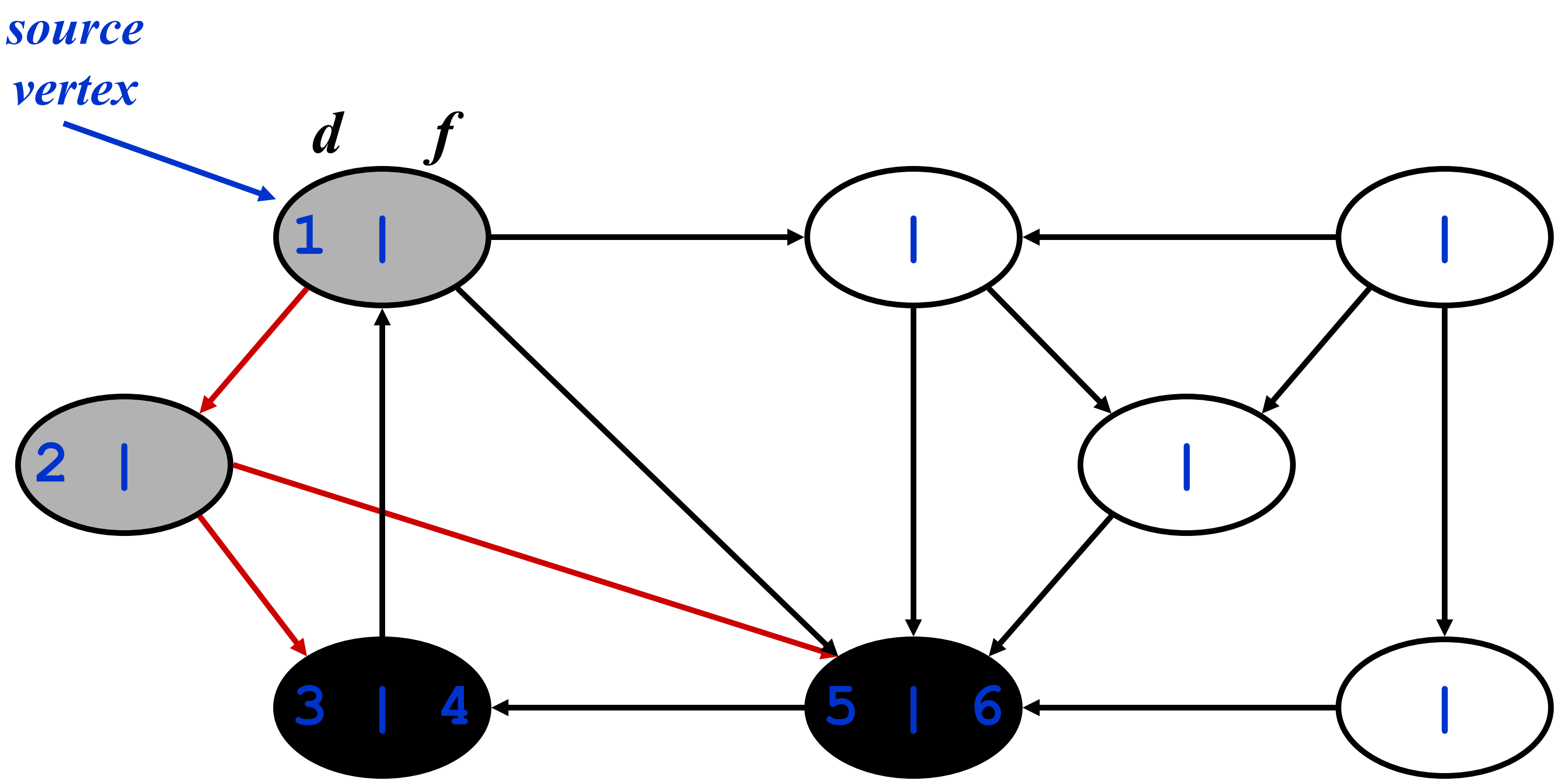
# DFS Example



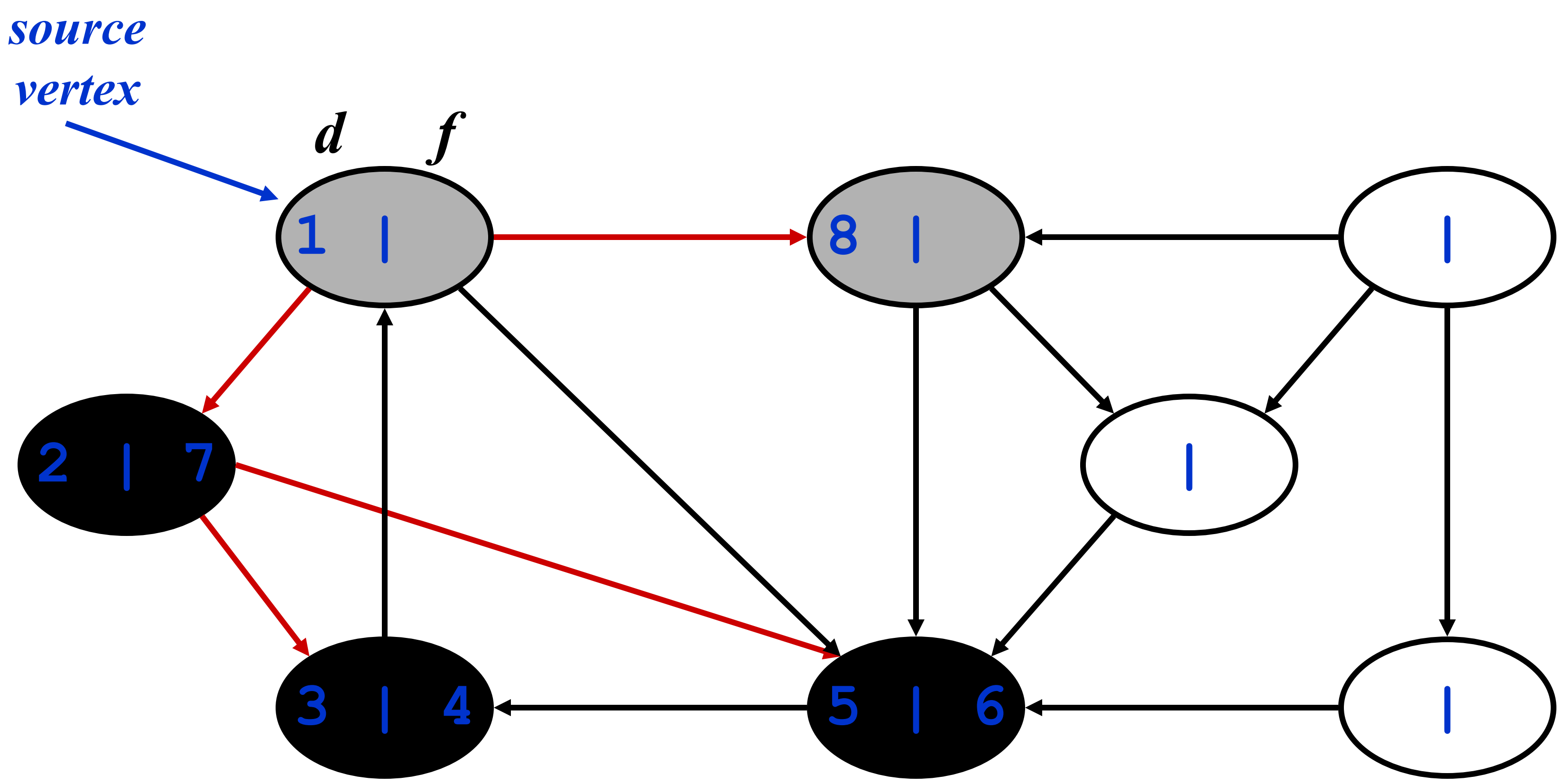
# DFS Example



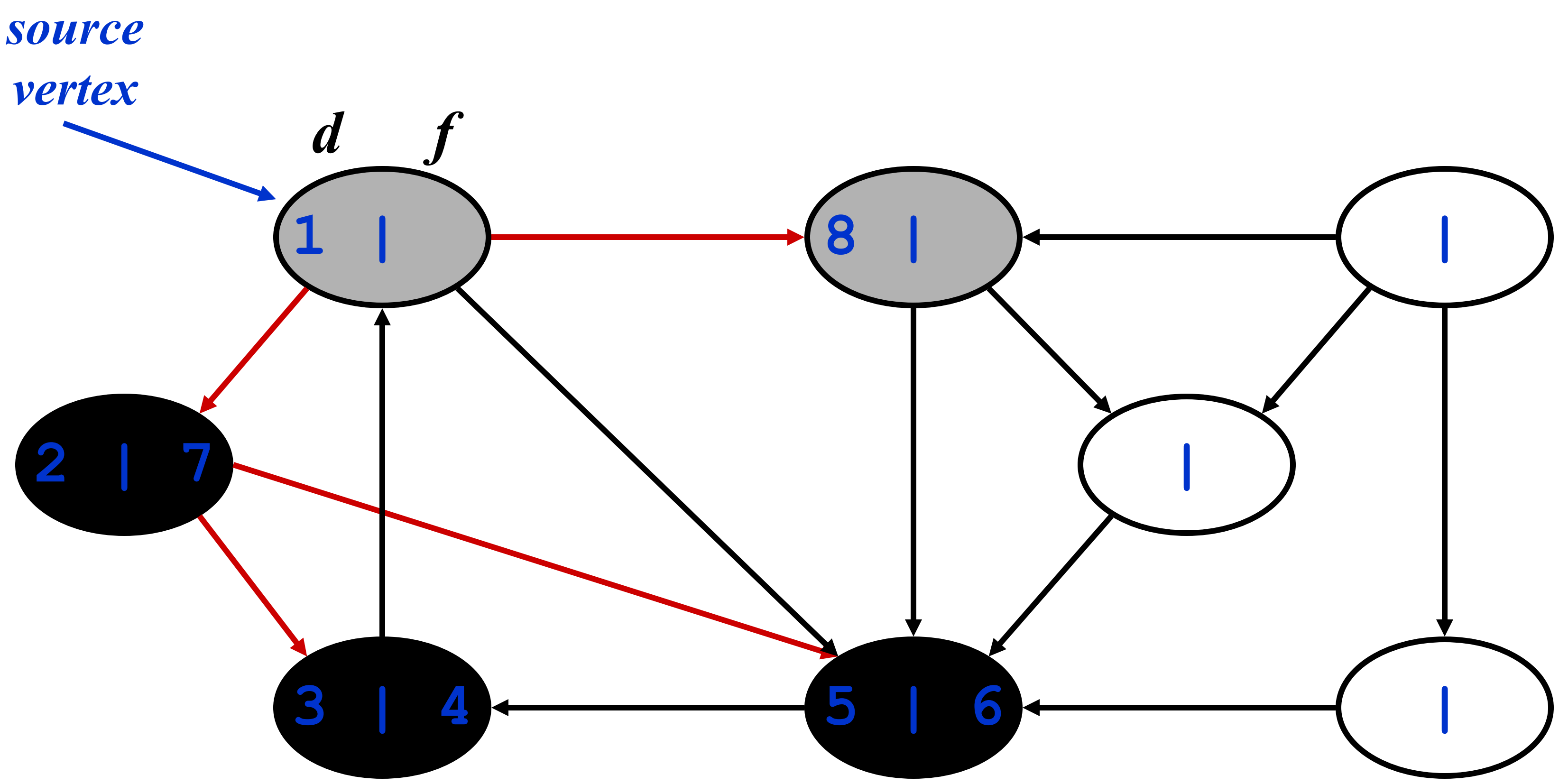
# DFS Example



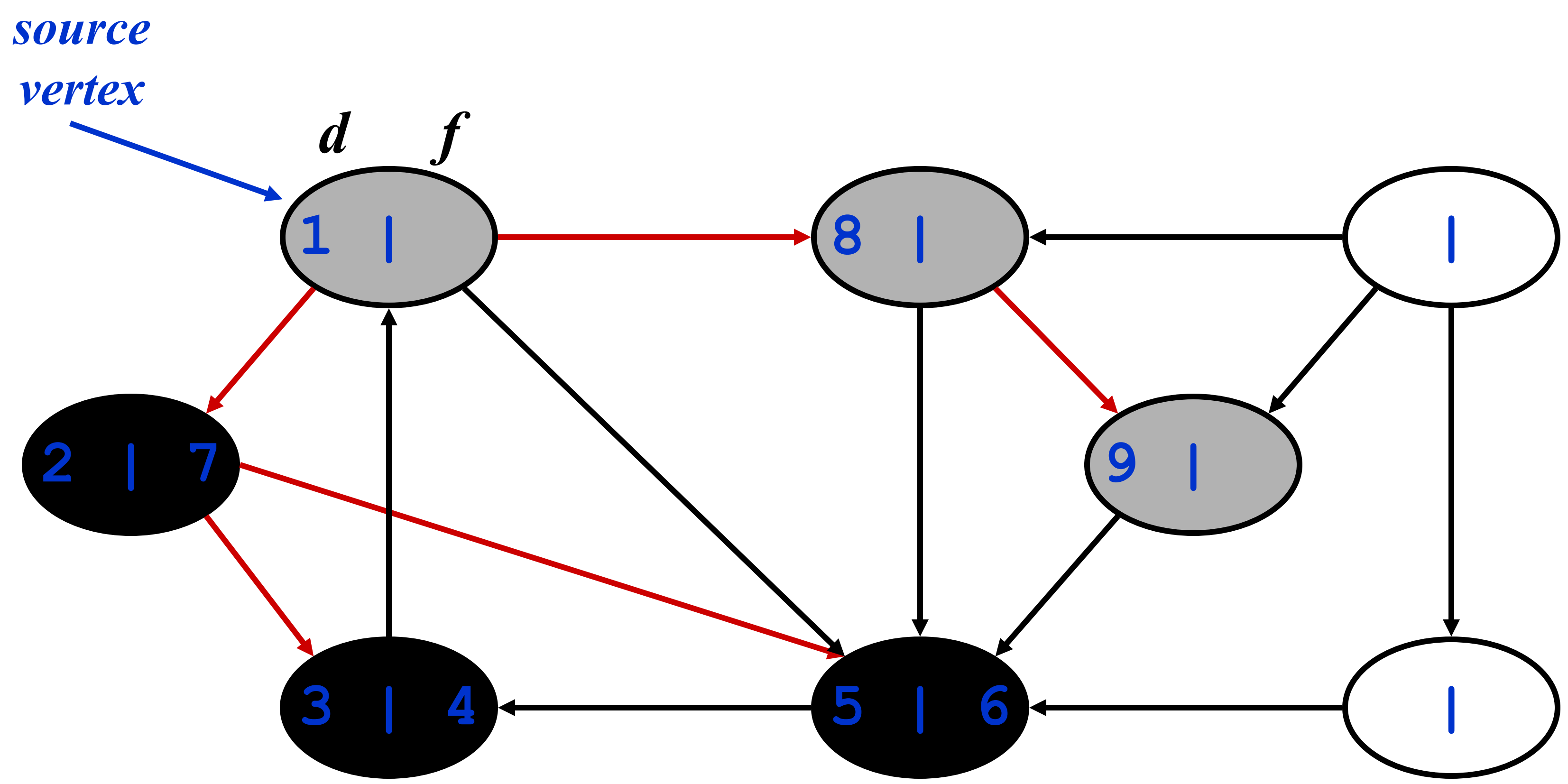
# DFS Example



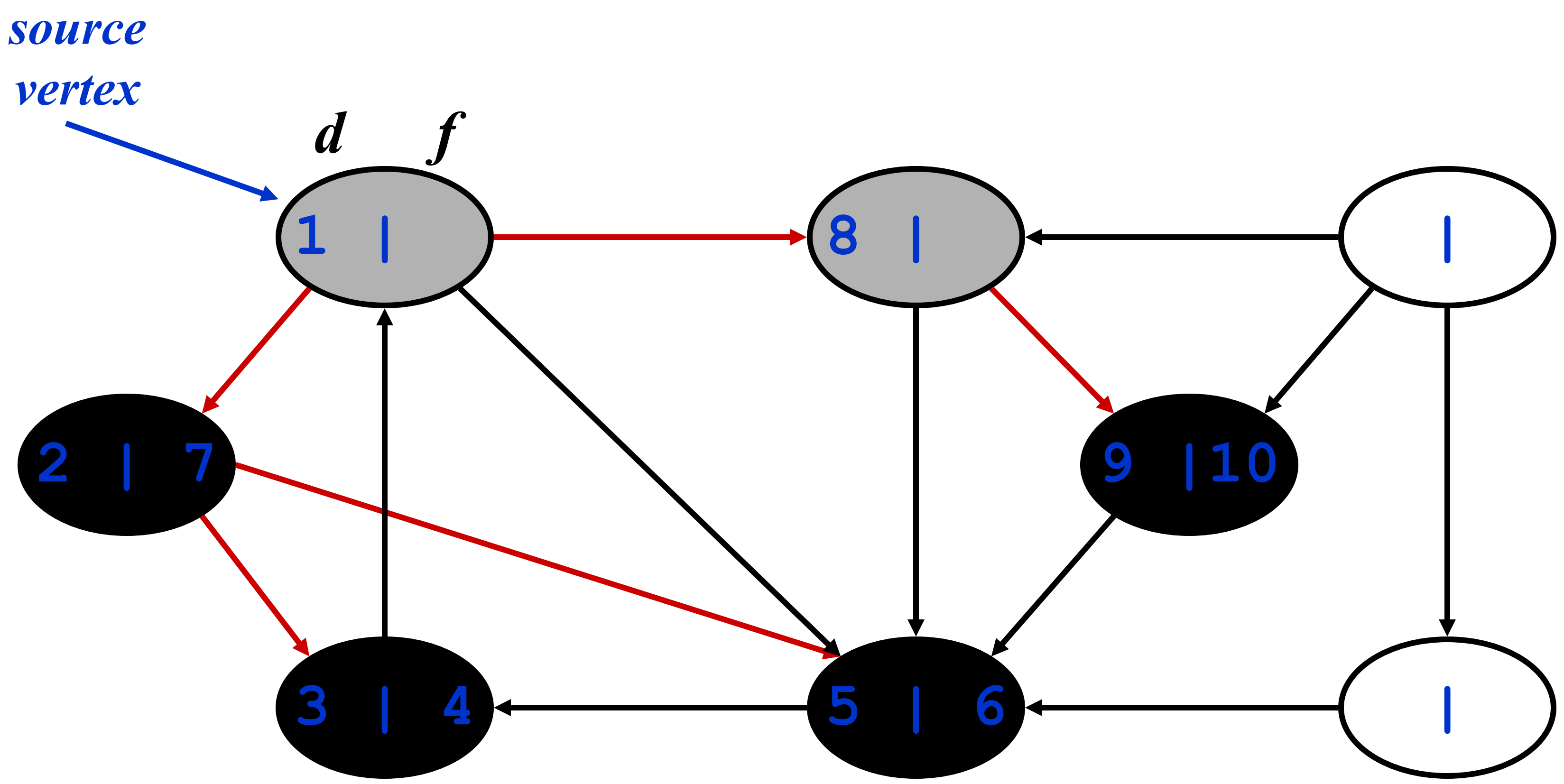
# DFS Example



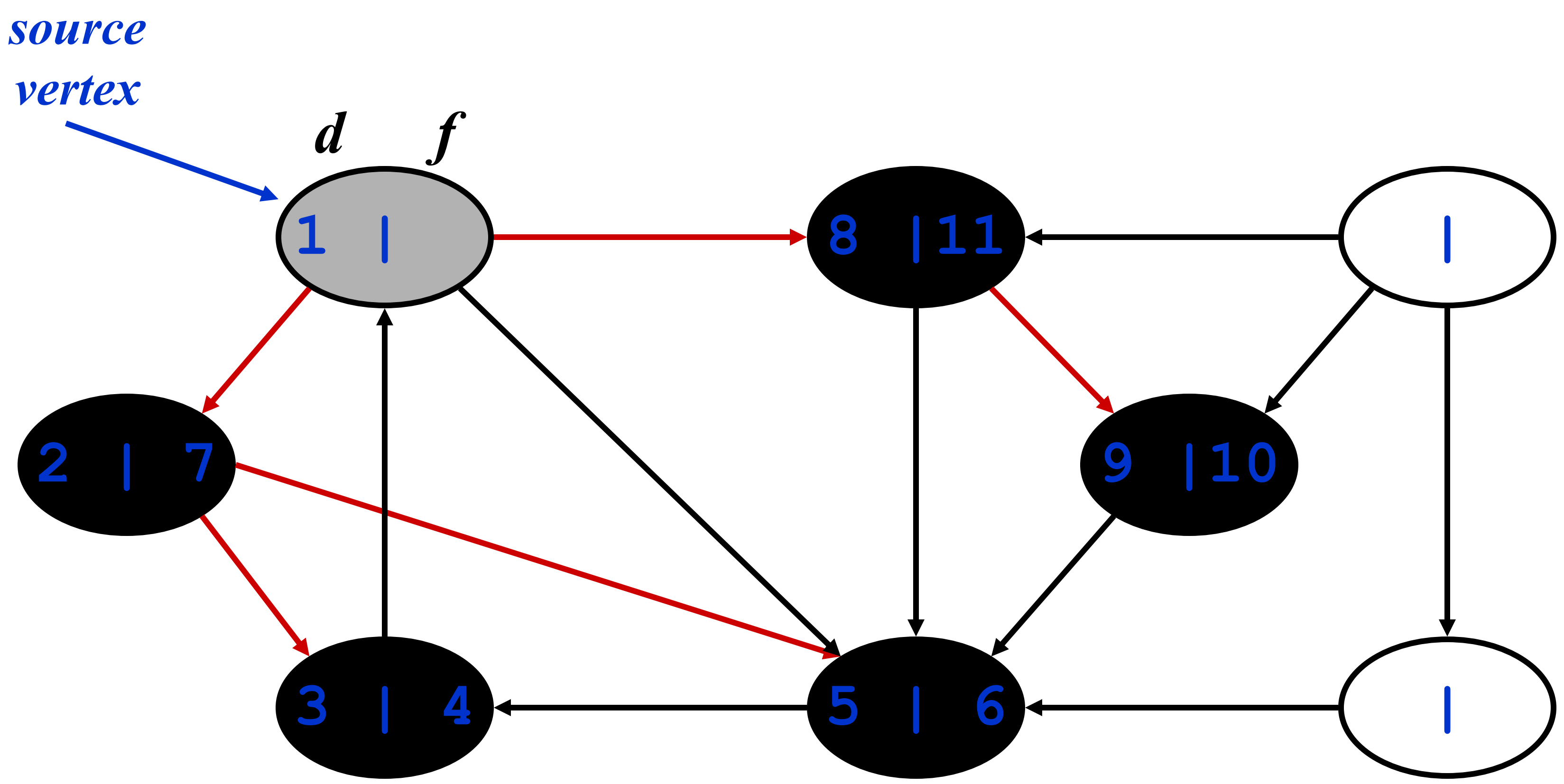
# DFS Example



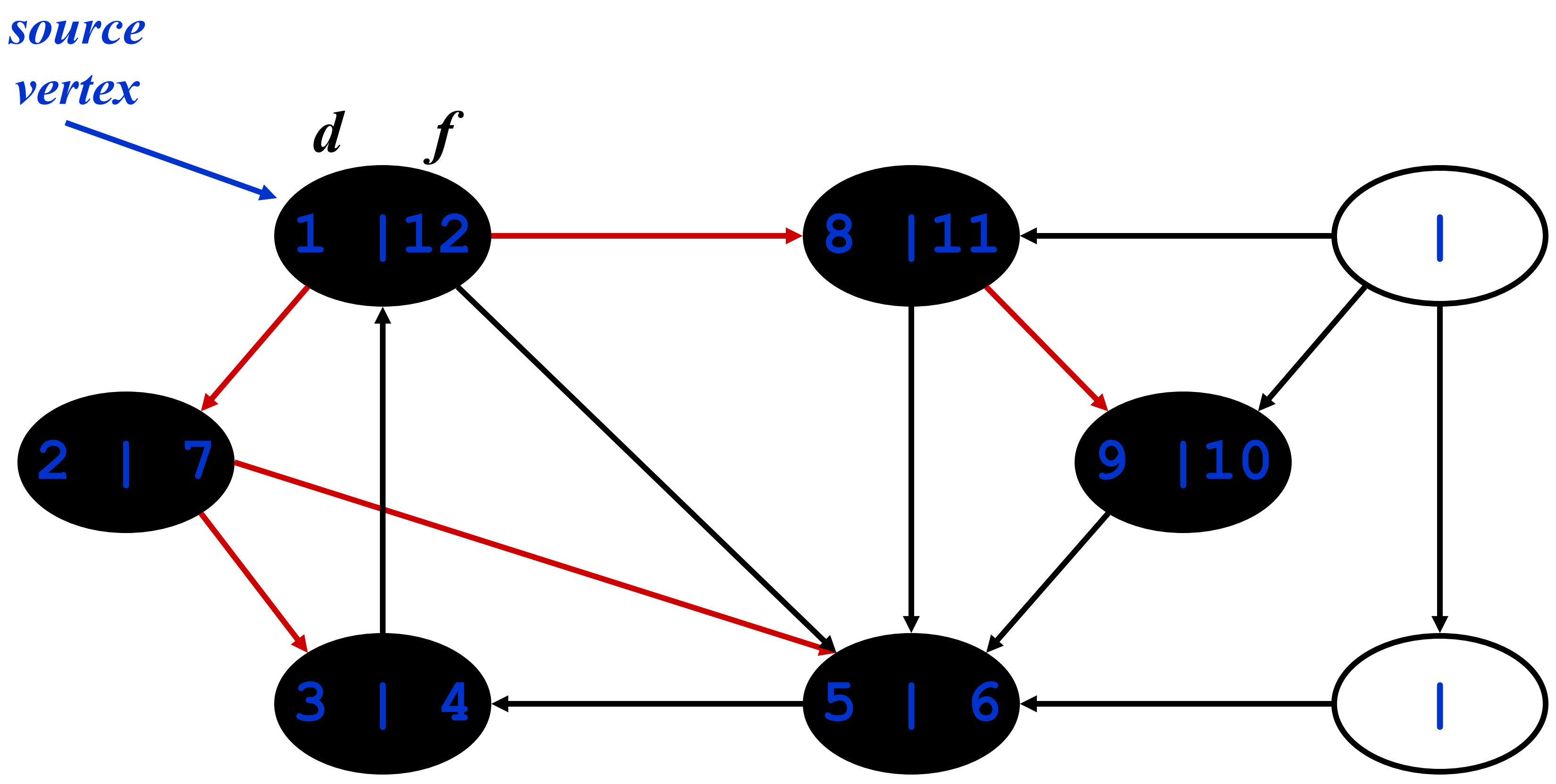
# DFS Example



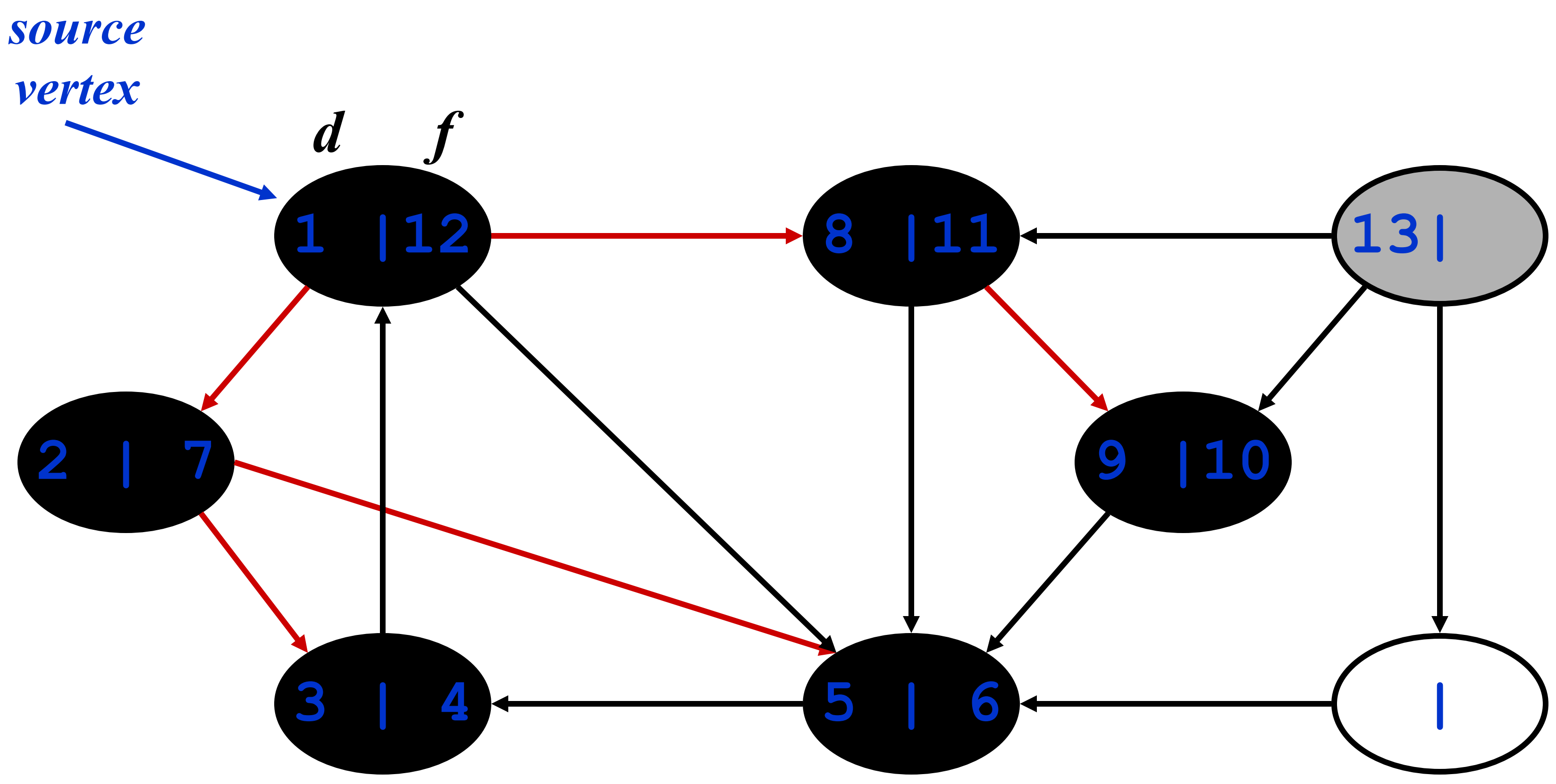
# DFS Example



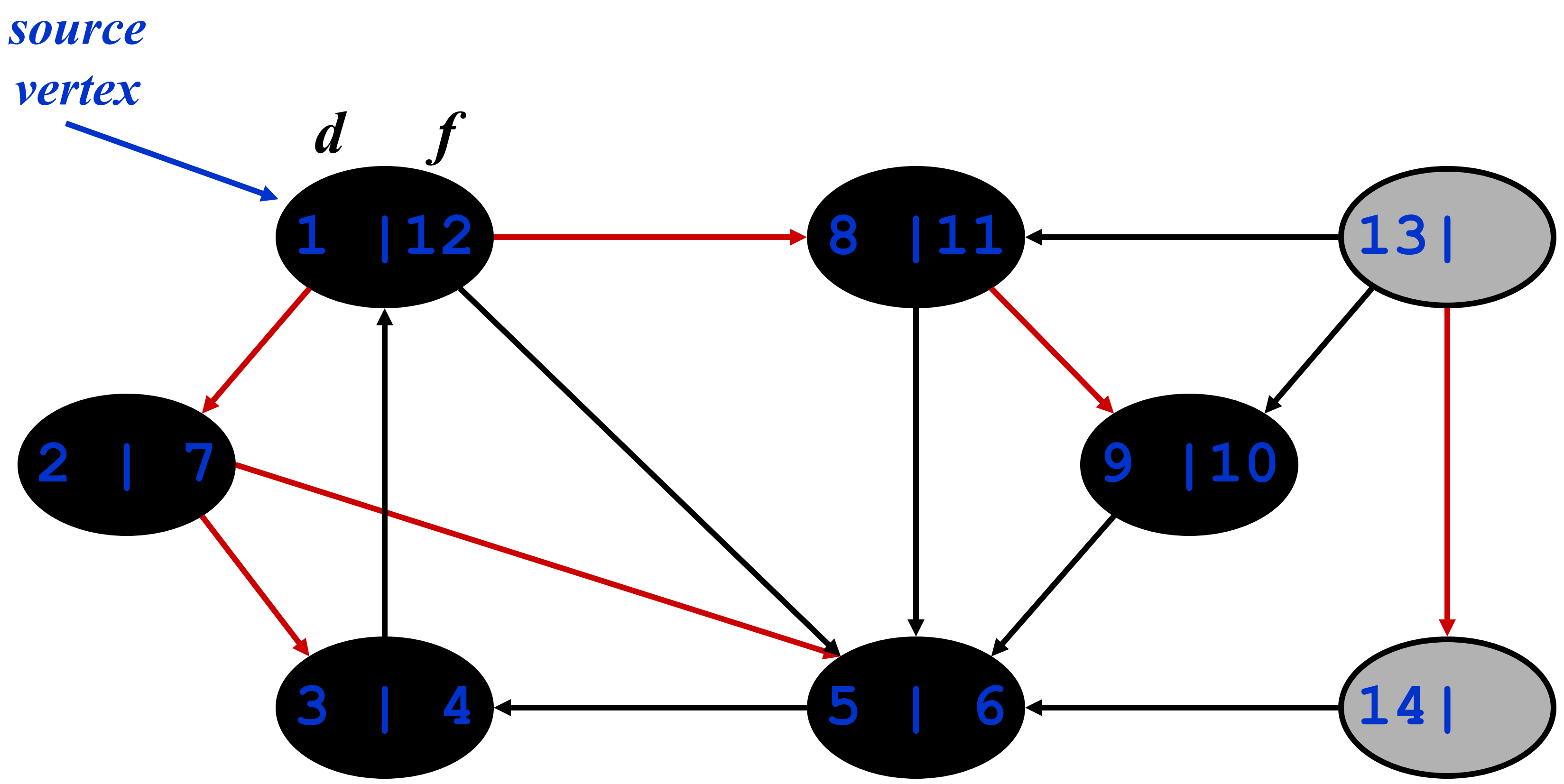
# DFS Example



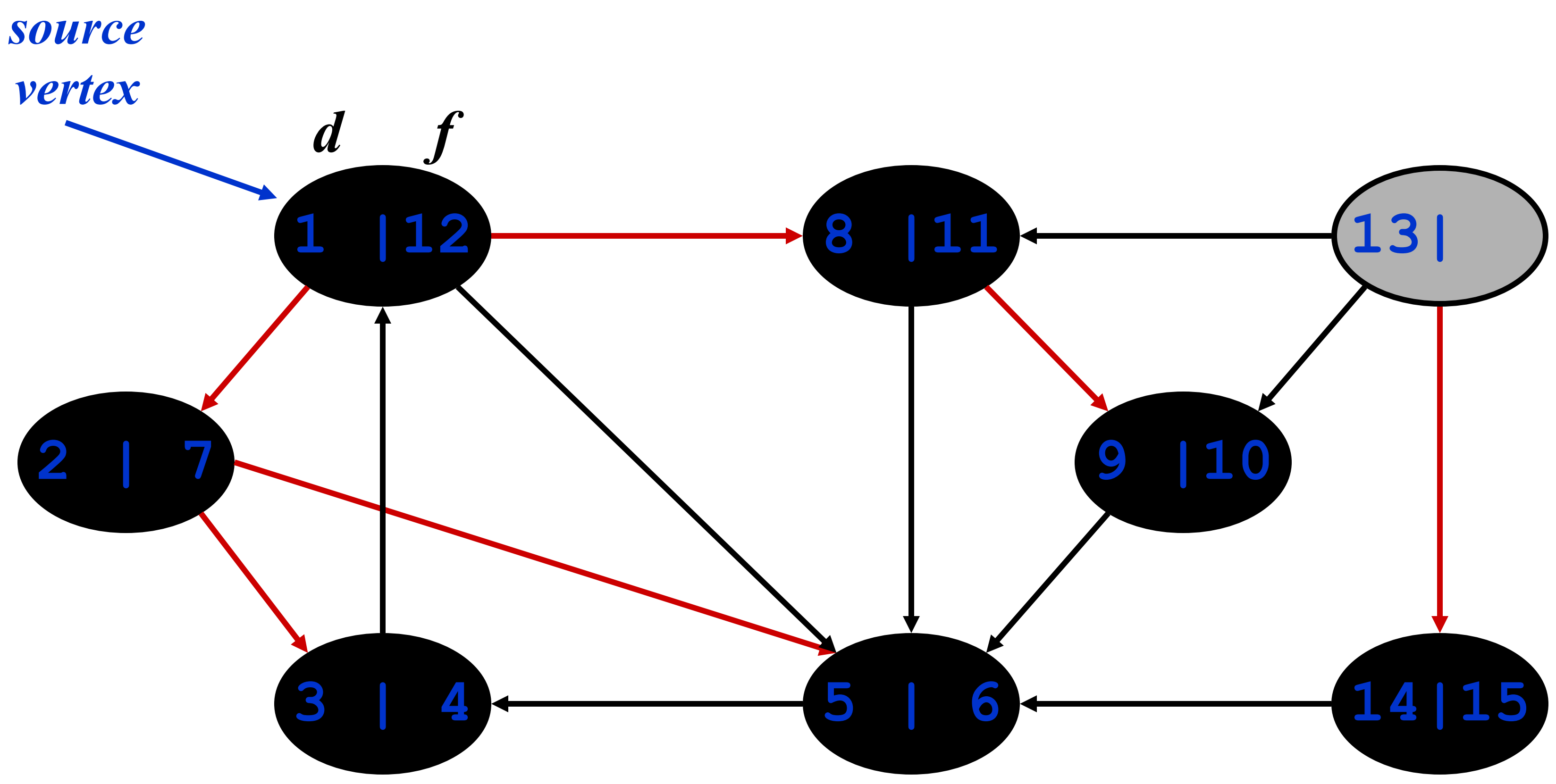
# DFS Example



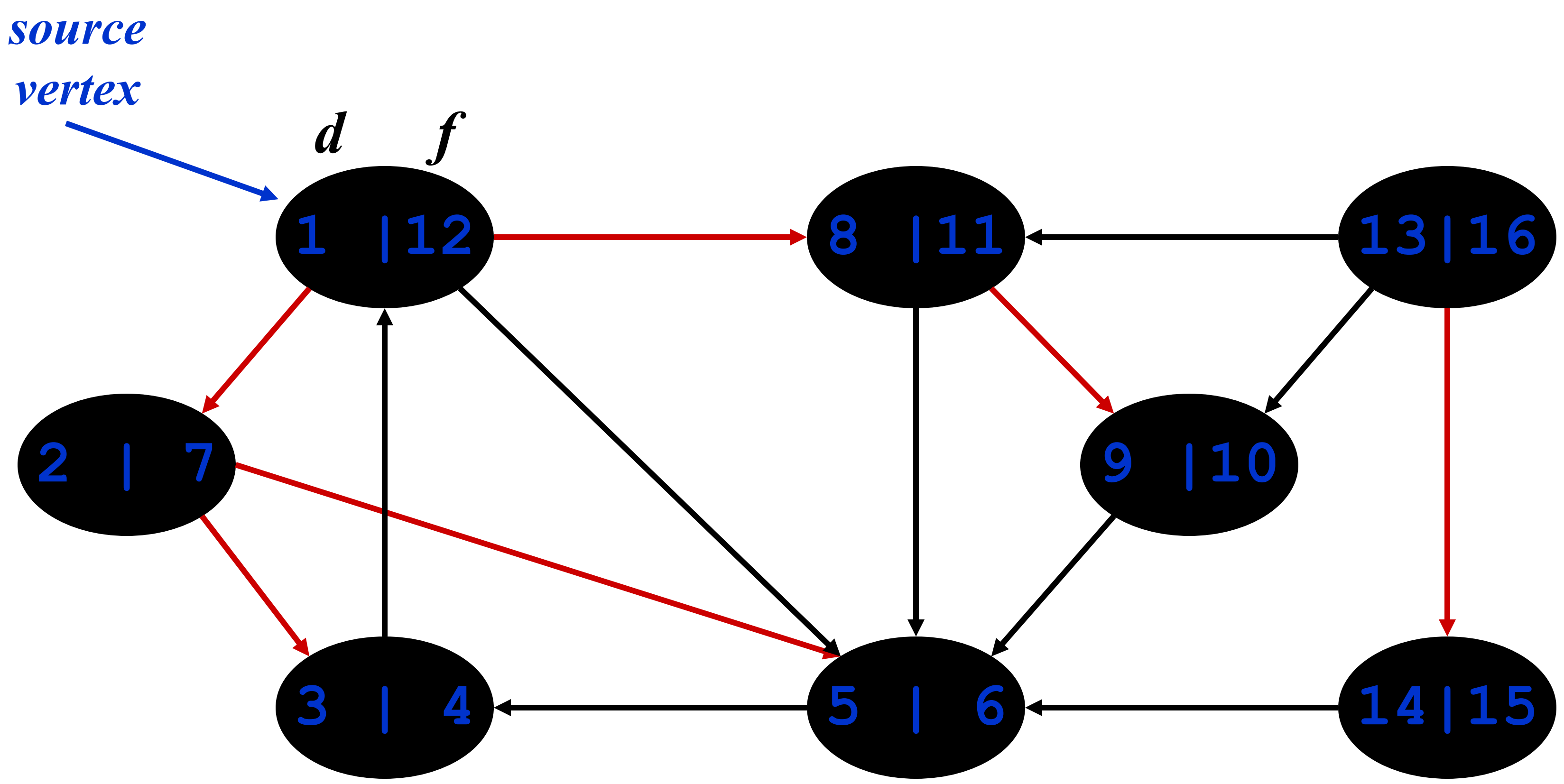
# DFS Example



# DFS Example



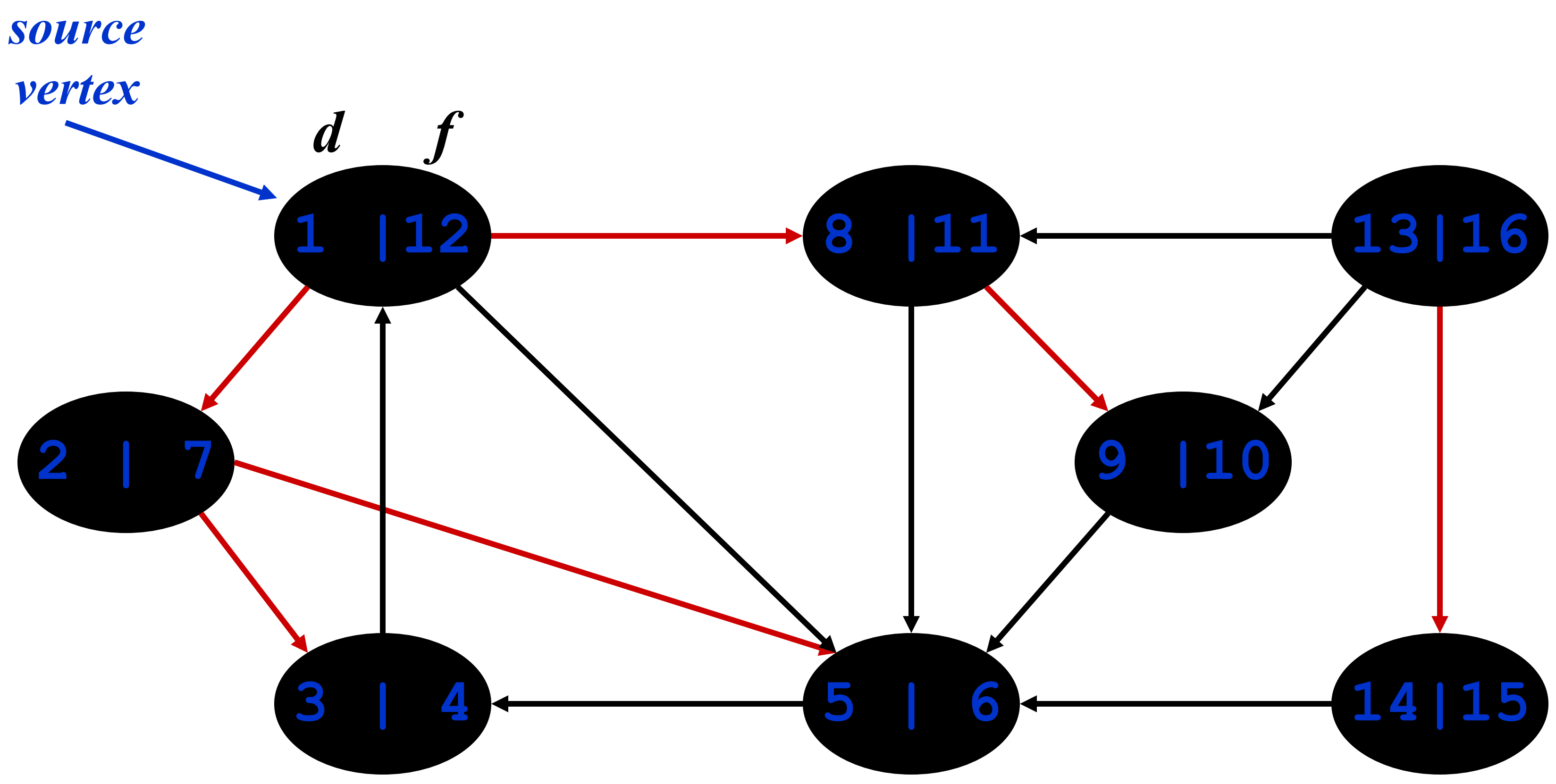
# DFS Example



# DFS: Kinds of edges

- DFS introduces an important distinction among edges in the original graph:
  - *Tree edge*: encounter new (white) vertex

# DFS Example

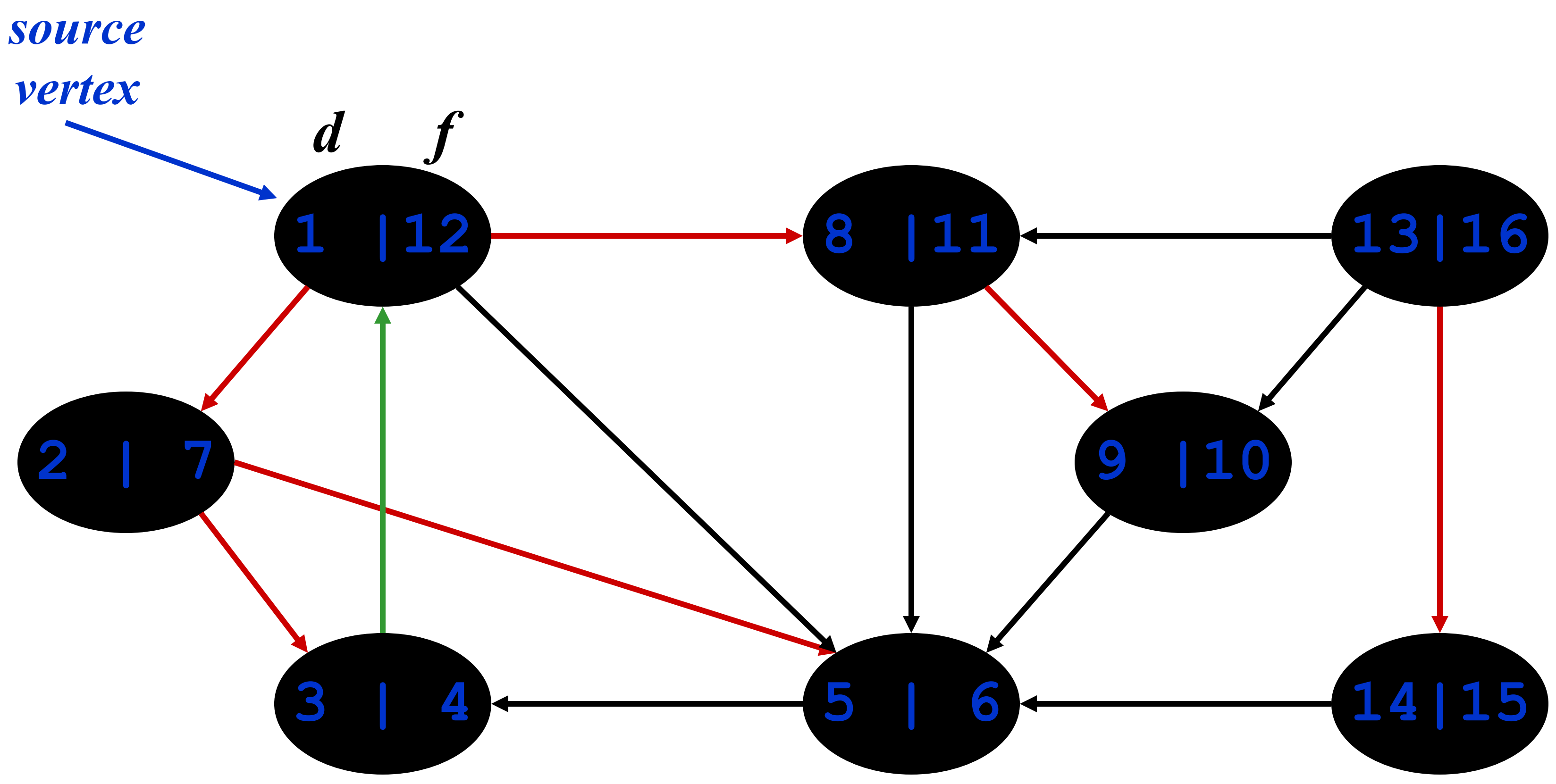


*Tree edges*

## DFS: Kinds of edges

- DFS introduces an important distinction among edges in the original graph:
  - *Tree edge*: encounter new (white) vertex
  - *Back edge*: from descendent to ancestor
    - Encounter a grey vertex (grey to grey)

# DFS Example

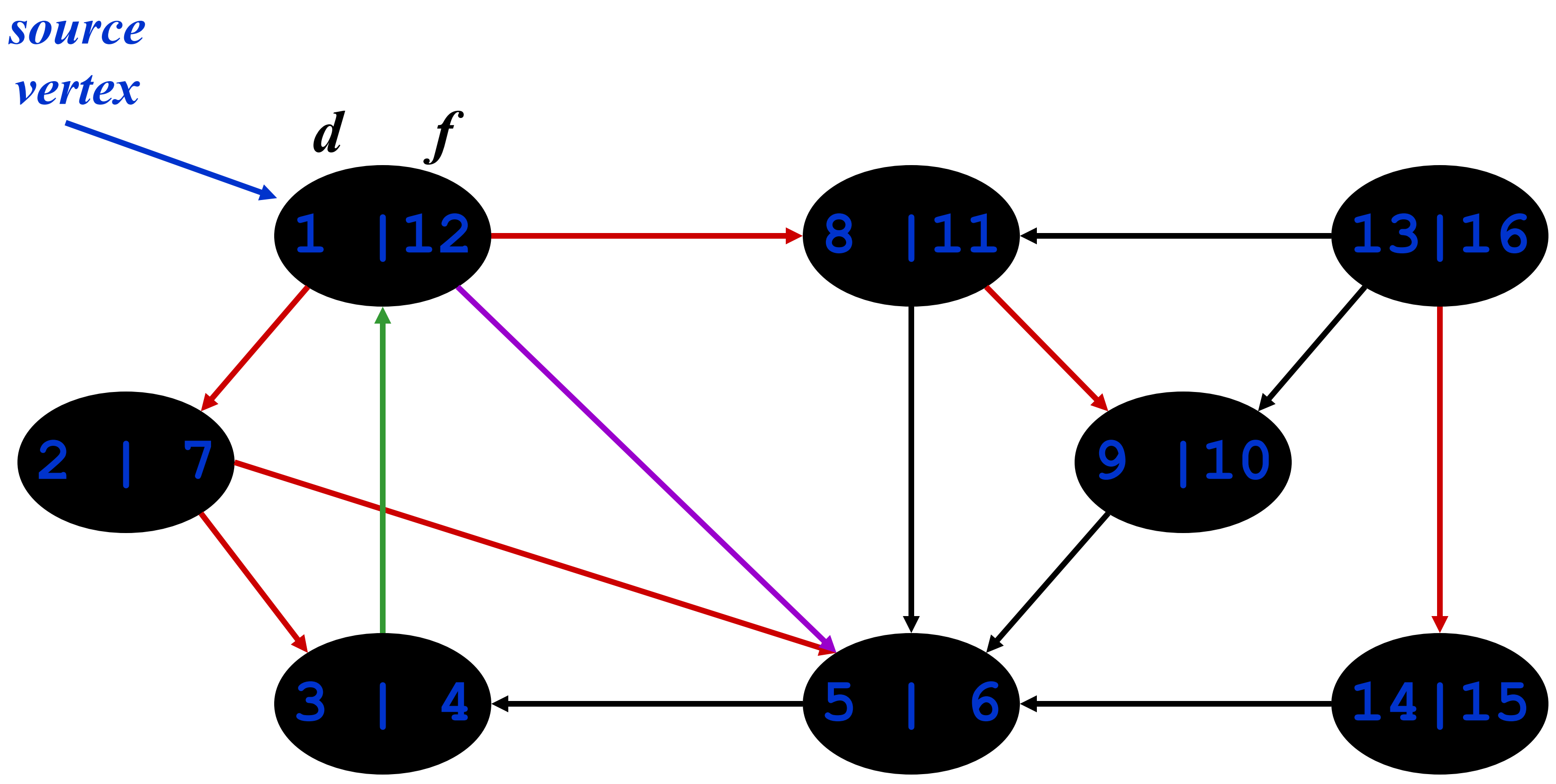


*Tree edges* *Back edges*

## DFS: Kinds of edges

- DFS introduces an important distinction among edges in the original graph:
  - *Tree edge*: encounter new (white) vertex
  - *Back edge*: from descendent to ancestor
  - *Forward edge*: from ancestor to descendent
    - Not a tree edge, though
    - From grey node to black node

# DFS Example

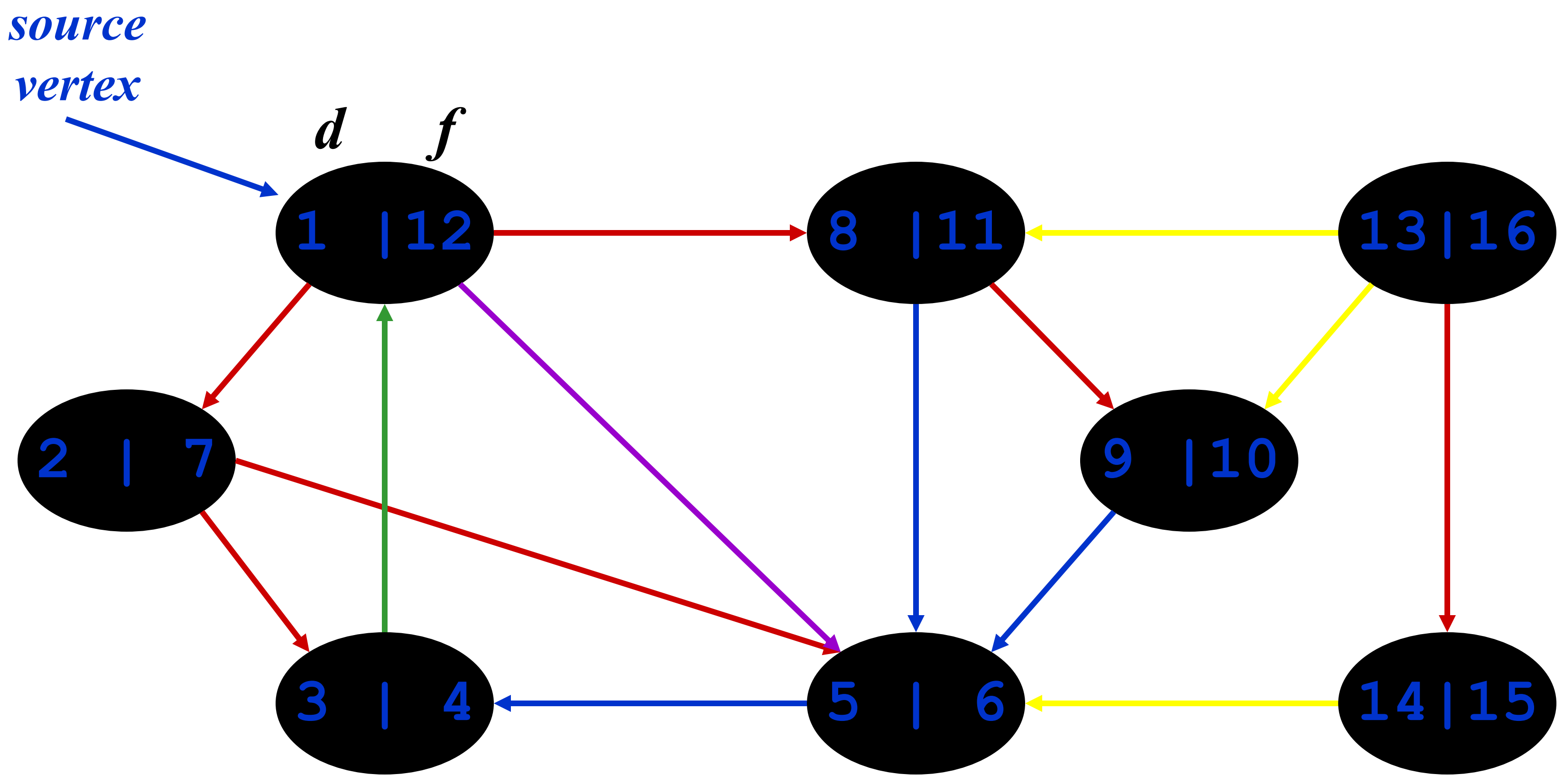


*Tree edges* *Back edges* *Forward edges*

# DFS: Kinds of edges

- DFS introduces an important distinction among edges in the original graph:
  - *Tree edge*: encounter new (white) vertex
  - *Back edge*: from descendent to ancestor
  - *Forward edge*: from ancestor to descendent
  - *Cross edge*: between a tree or subtrees
    - From a grey node to a black node

# DFS Example



*Tree edges* *Back edges* *Forward edges* *Cross edges*

## DFS: Kinds of edges

- DFS introduces an important distinction among edges in the original graph:
  - *Tree edge*: encounter new (white) vertex
  - *Back edge*: from descendent to ancestor
  - *Forward edge*: from ancestor to descendent
  - *Cross edge*: between a tree or subtrees
- Note: tree & back edges are important; most algorithms don't distinguish forward & cross

## Review: Dynamic Programming

- Summary of the basic idea:
  - Optimal substructure: optimal solution to problem consists of optimal solutions to subproblems
  - Overlapping subproblems: few subproblems in total, many recurring instances of each
  - Solve bottom-up, building a table of solved subproblems that are used to solve larger ones

## Greedy Algorithms

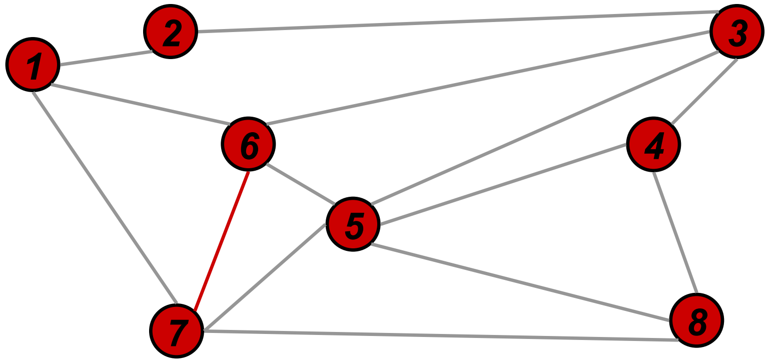
- Like dynamic programming, used to solve optimization problems.
- Problems exhibit optimal substructure (like DP).
- Problems also exhibit the **greedy-choice** property.
  - When we have a choice to make, make the one that looks best *right now*.
  - Make a **locally optimal choice** in hope of getting a **globally optimal solution**.

## Greedy Algorithms

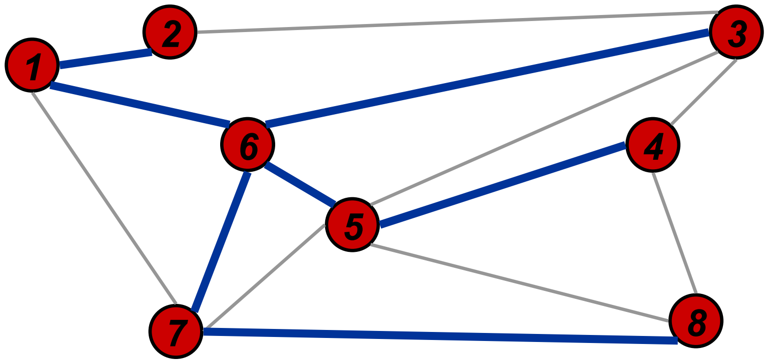
- A *greedy algorithm* always makes the choice that looks best at the moment
  - The hope: a locally optimal choice will lead to a globally optimal solution
  - For some problems, it works
- Dynamic programming can be overkill; greedy algorithms tend to be easier to code

# Spanning Tree

- Spanning tree. Let T = (V, F) be a subgraph of G = (V, E).
- T is a spanning tree of G: *T that contains all the vertices* of a graph *G*.
  - T is acyclic and connected.
  - T is connected and has |V| - 1 arcs.
  - T is acyclic and has |V| - 1 arcs.
  - T is minimally connected: removal of any arc disconnects it.
  - T is maximally acyclic: addition of any arc creates a cycle.
  - T has a unique simple path between every pair of vertices.



G = (V, E)



T = (V, F)

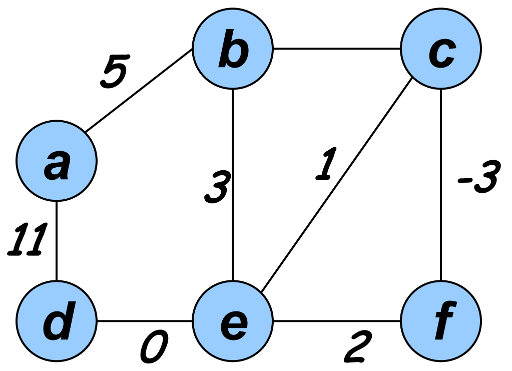
# Minimum Spanning Trees

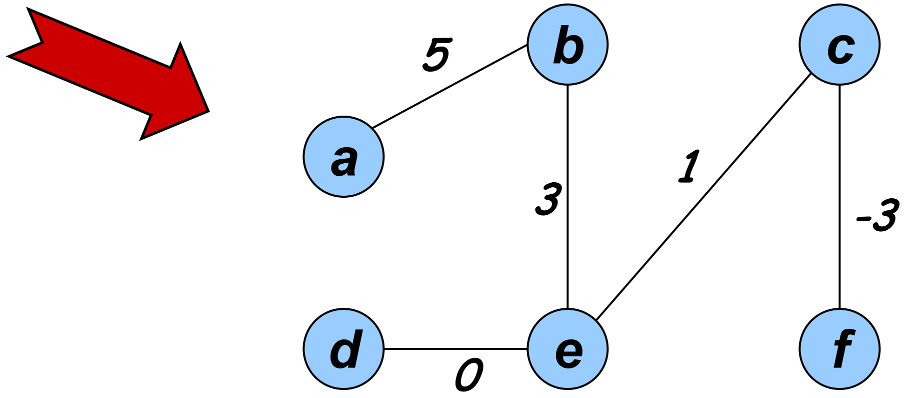
•*A spanning tree for G is a free tree that connects all the vertices in V.*

•*The cost of a spanning tree is the sum of the costs of the edges in the tree.*

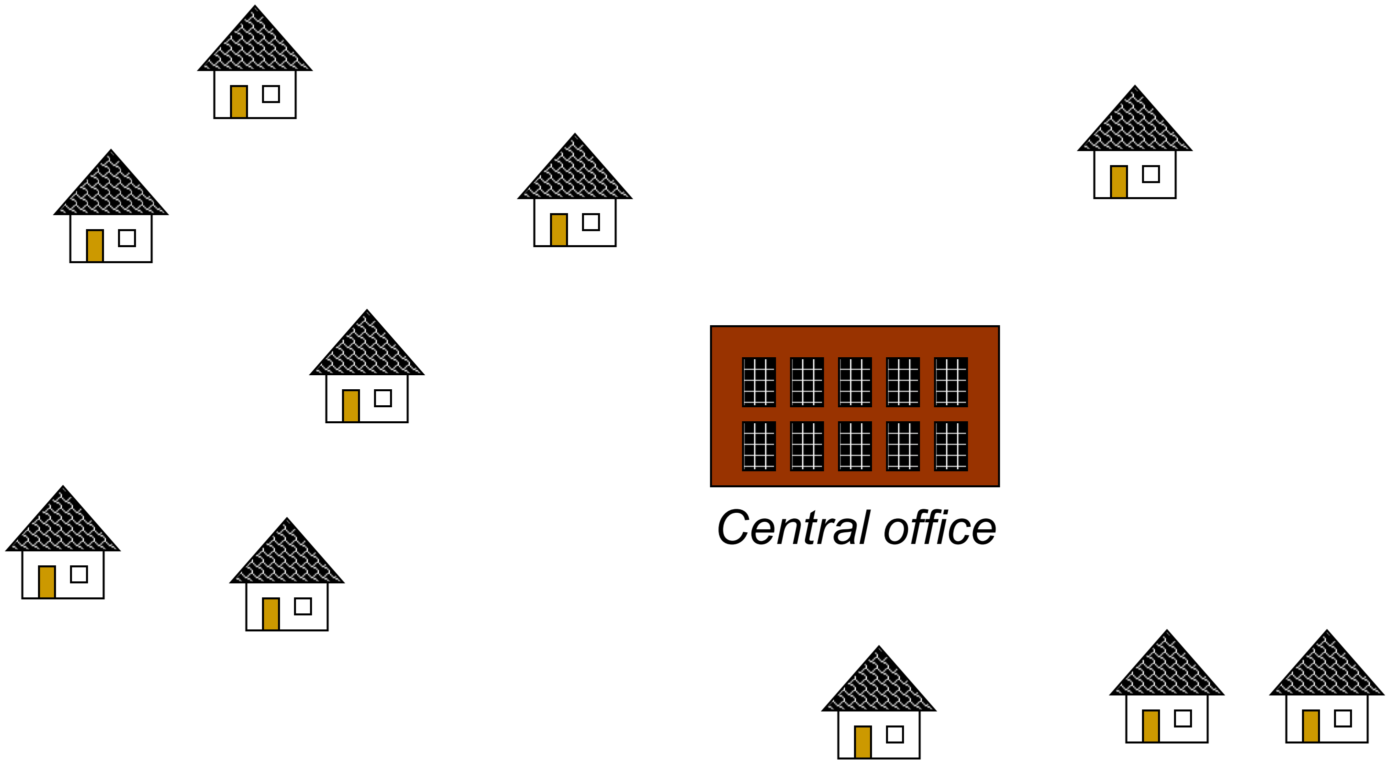
•***Given:*** *a Connected, undirected, weighted graph, G =(V, E) in which each edge (u, v) in E has a cost c (u, v) attached to it.*

•***Find:*** *Minimum - weight spanning tree, T*

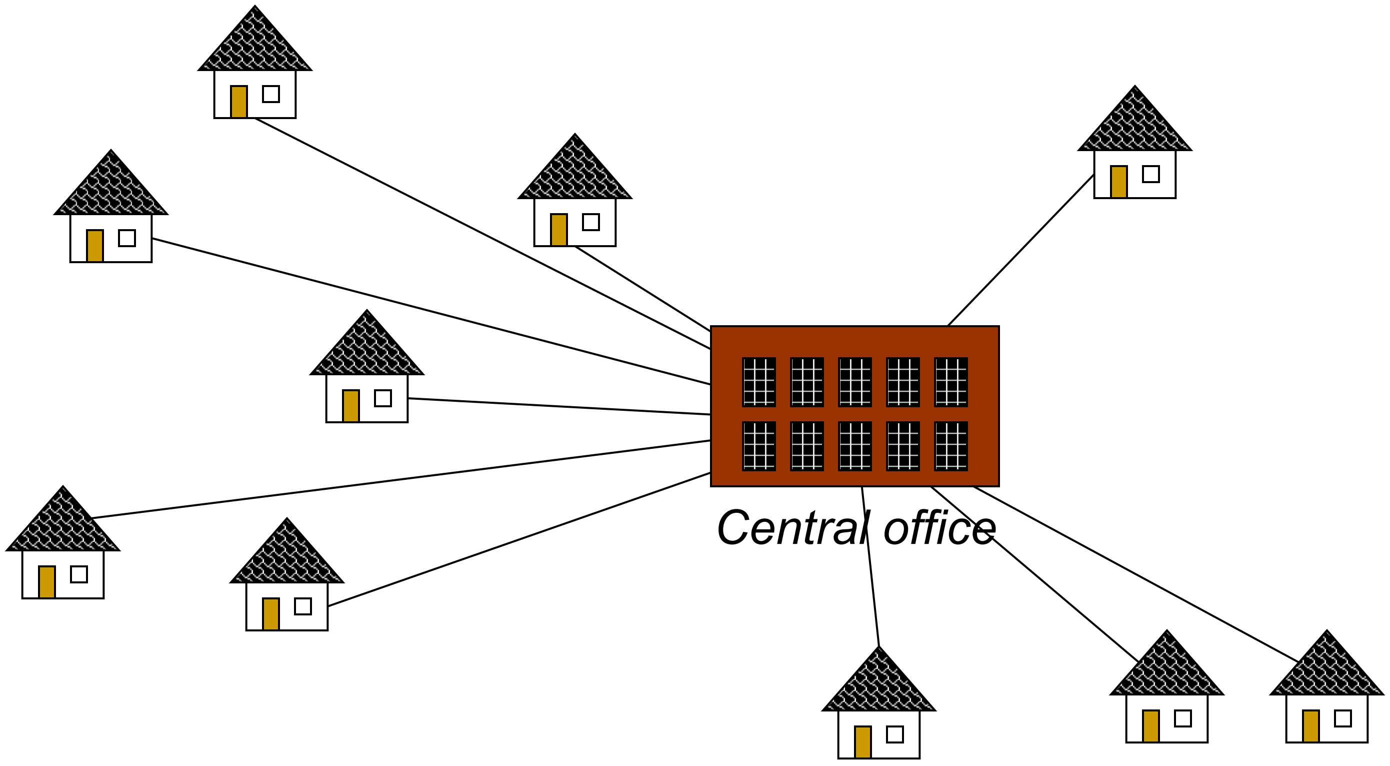




## Problem: Laying Telephone Wire

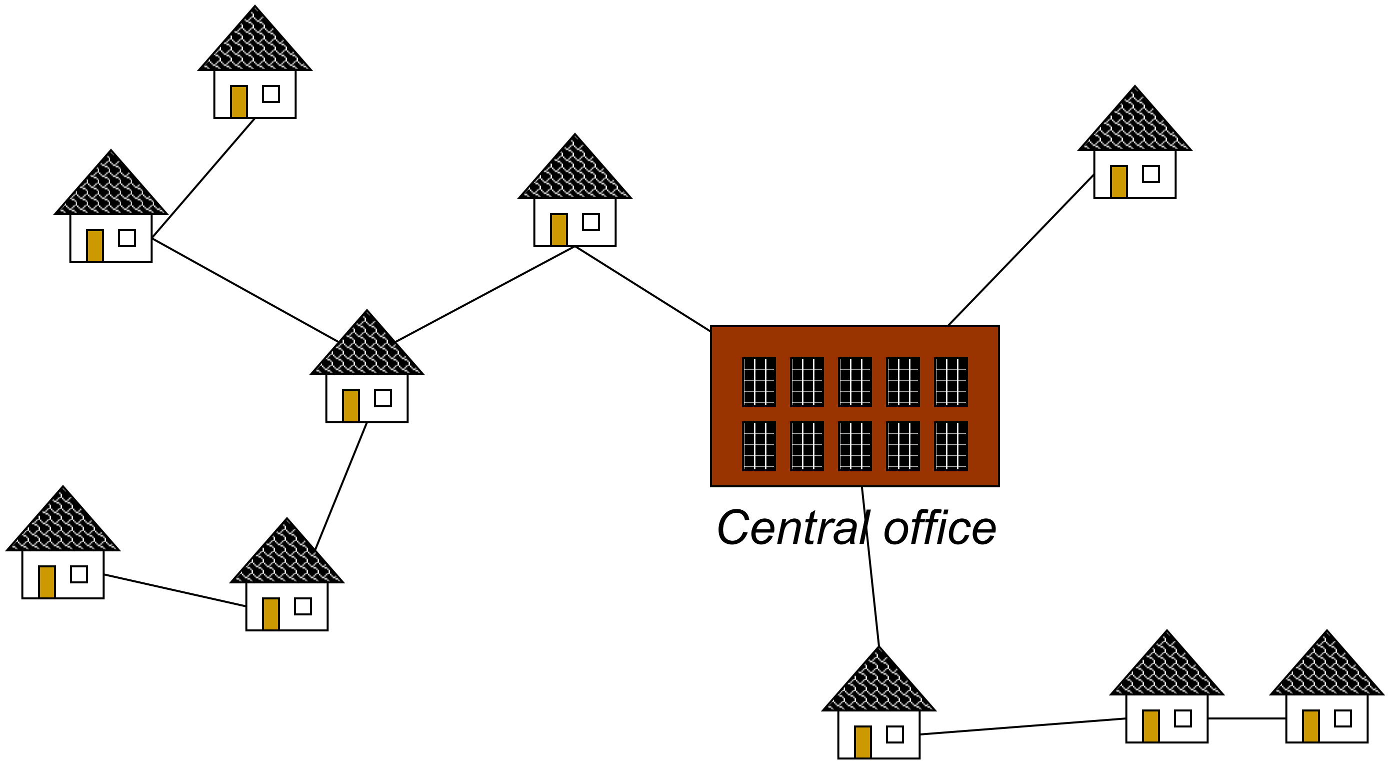


# Wiring: Naïve Approach



***Expensive!***

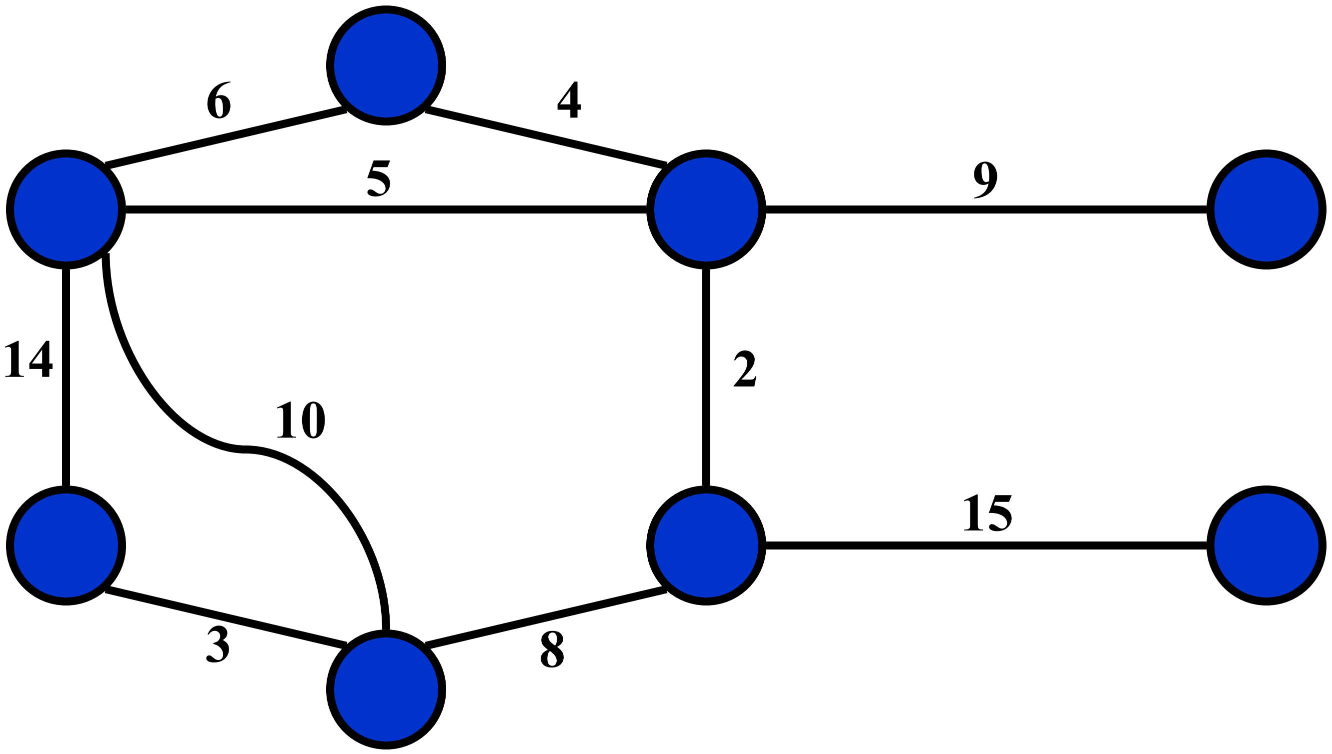
# Wiring: Better Approach



*Minimize the total length of wire connecting the customers*

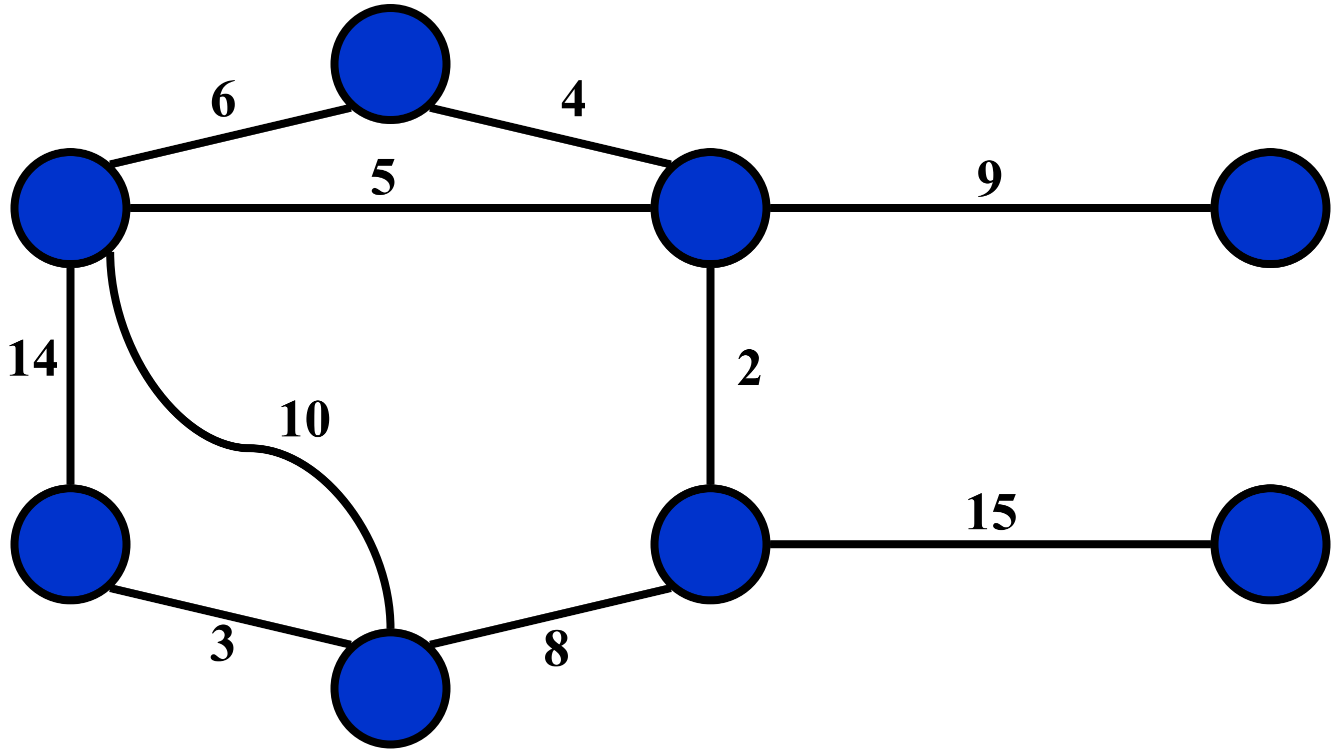
## Minimum Spanning Tree

- Problem: given a connected, undirected, weighted graph:



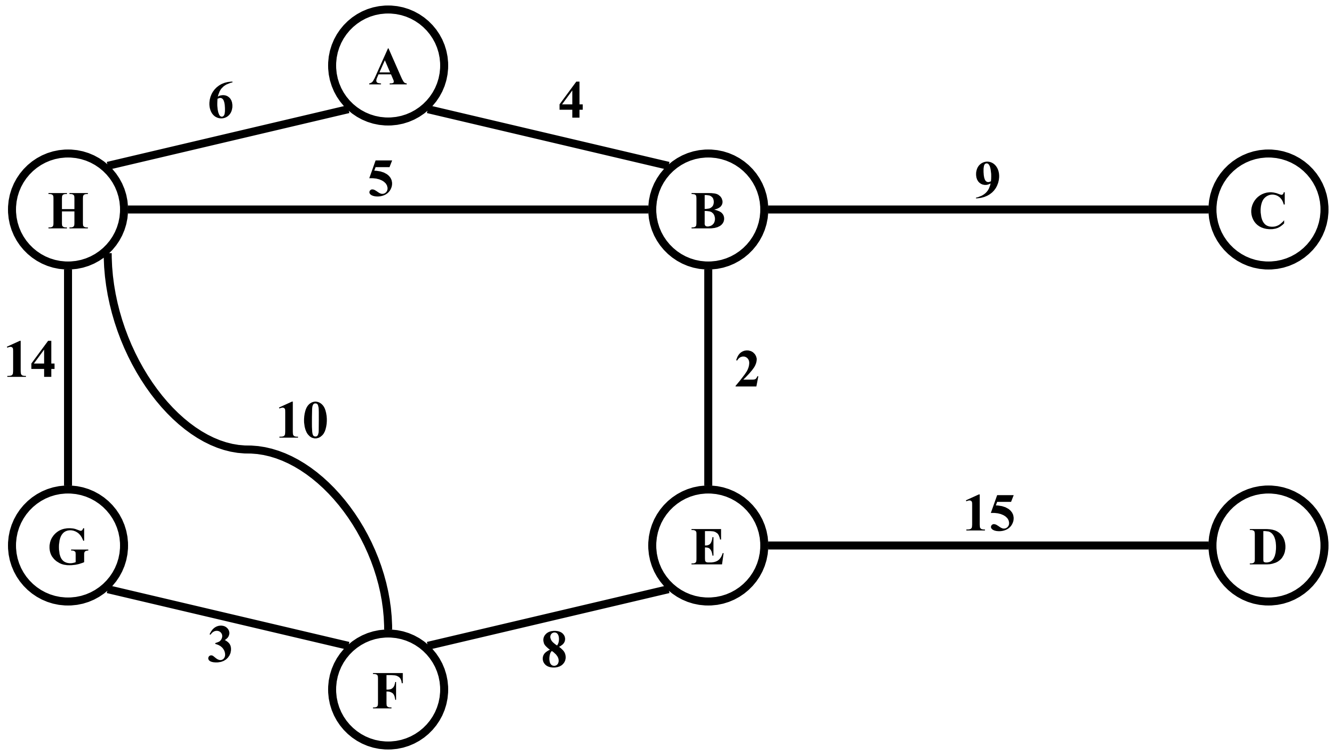
## Minimum Spanning Tree

- Problem: given a connected, undirected, weighted graph, find a *spanning tree* using edges that minimize the total weight



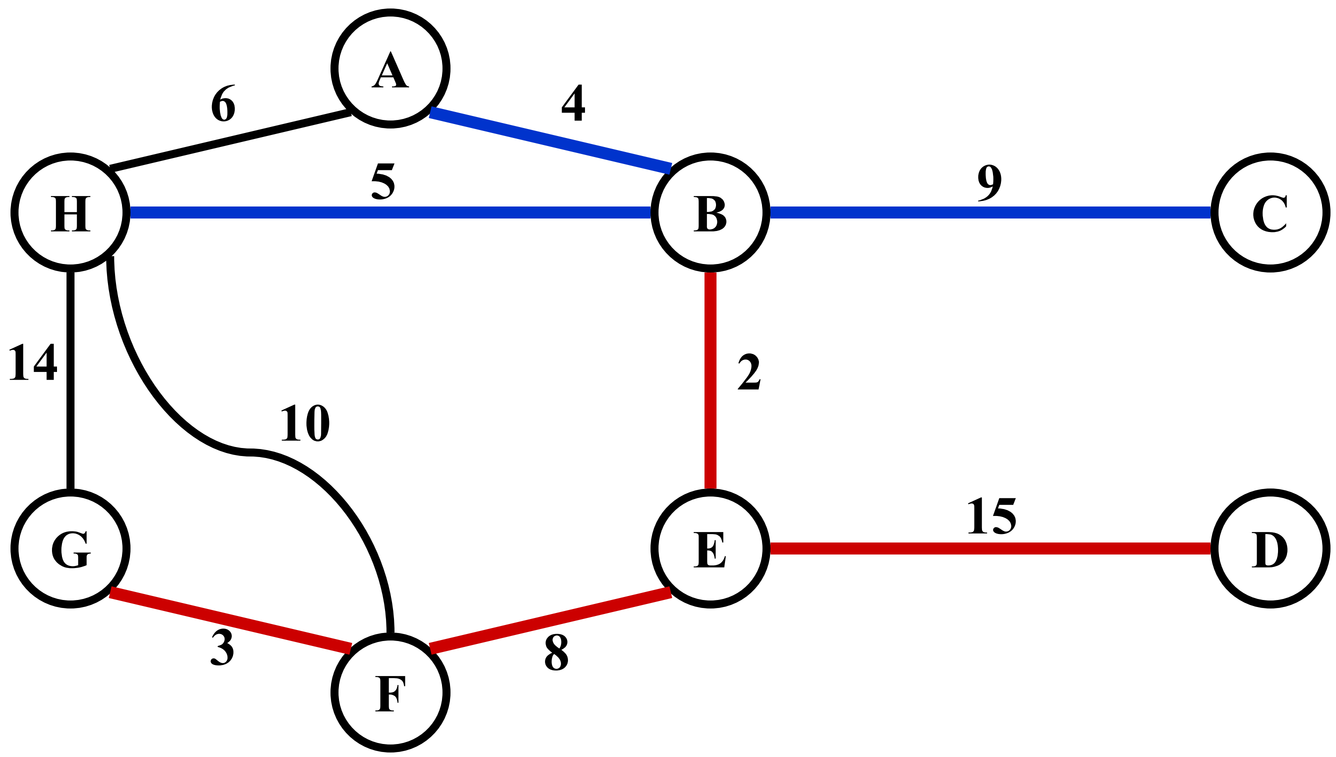
## Minimum Spanning Tree

- *Which edges form the minimum spanning tree (MST) of the below graph?*



## Minimum Spanning Tree

- Answer:



## Minimum Spanning Tree

- MSTs satisfy the *optimal substructure* property: an optimal tree is composed of optimal subtrees
  - Let T be an MST of G with an edge $(u,v)$ in the middle
  - Removing $(u,v)$ partitions T into two trees $T_1$ and $T_2$
  - Claim: $T_1$ is an MST of $G_1 = (V_1,E_1)$, and $T_2$ is an MST of $G_2 = (V_2,E_2)$
  - Proof: $w(T) = w(u,v) + w(T_1) + w(T_2)$
    (There can't be a better tree than $T_1$ or $T_2$, or T would be suboptimal)

## Minimum Spanning Tree

- Thm:
  - Let T be MST of G, and let A $\subseteq$ T be subtree of T
  - Let $(u,v)$ be min-weight edge connecting A to V-A
  - Then $(u,v) \in$ T

## Minimum Spanning Tree

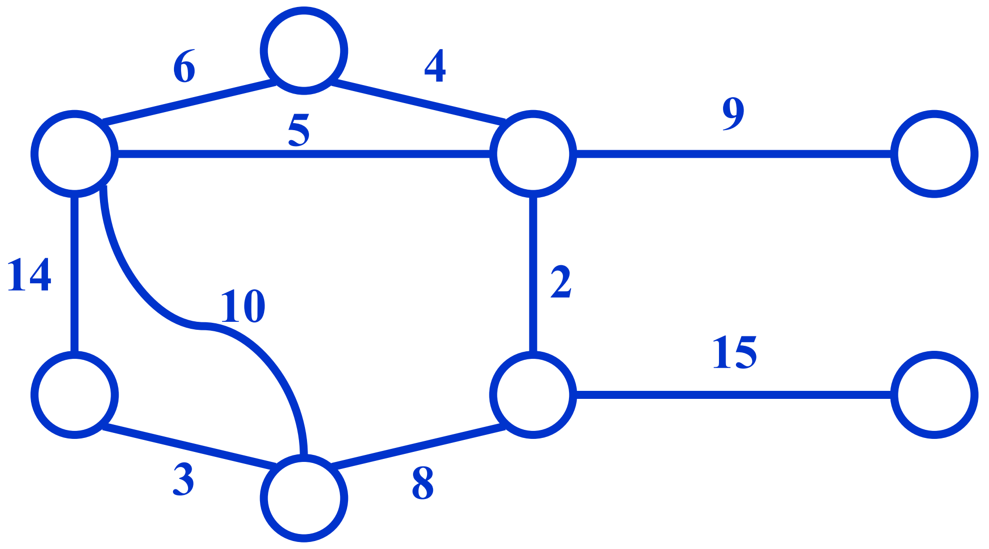
- Thm:
  - Let T be MST of G, and let A $\subseteq$ T be subtree of T
  - Let $(u,v)$ be min-weight edge connecting A to V-A
  - Then $(u,v) \in$ T

# Prim’s Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```

# Prim’s Algorithm

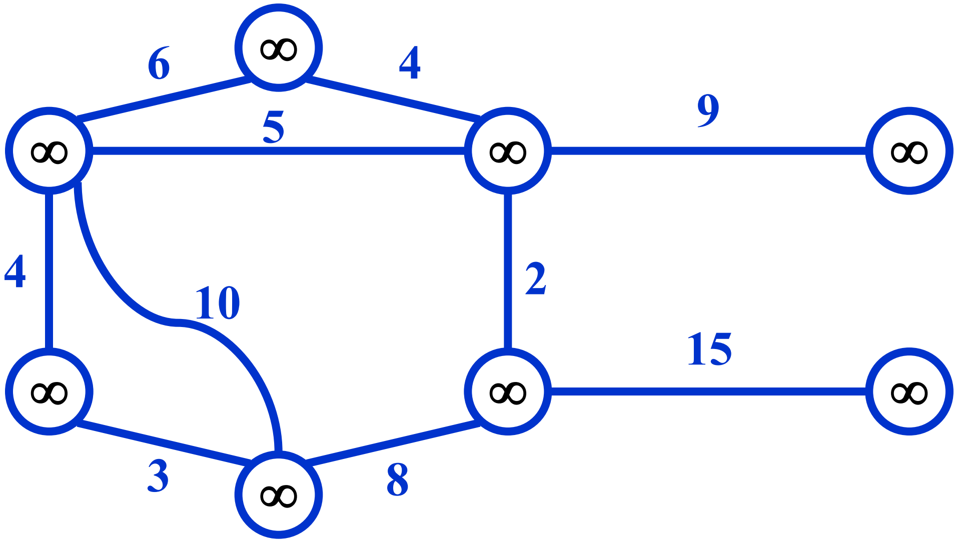
```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



*Run on example graph*

# Prim’s Algorithm

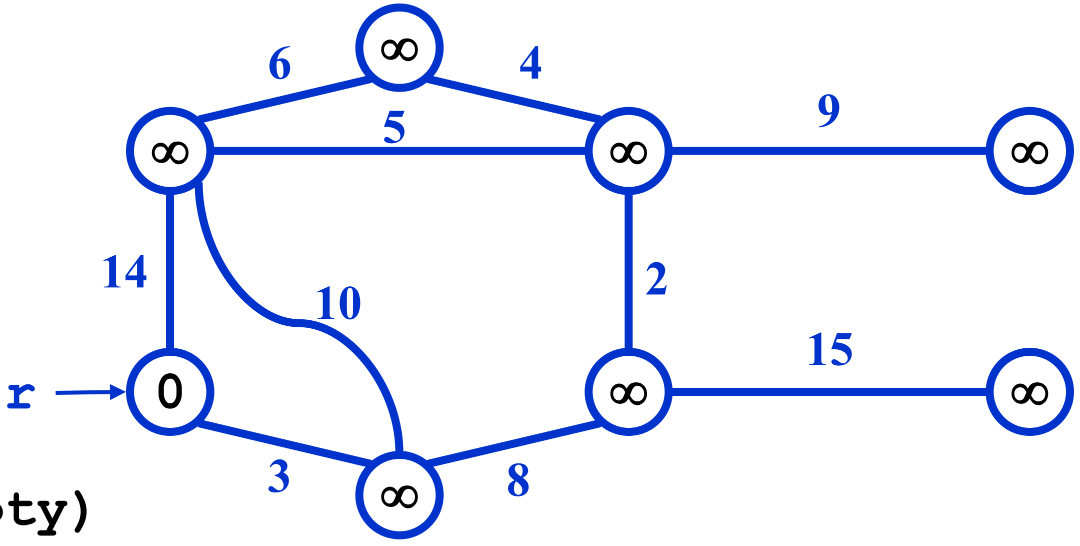
```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



*Run on example graph*

# Prim's Algorithm

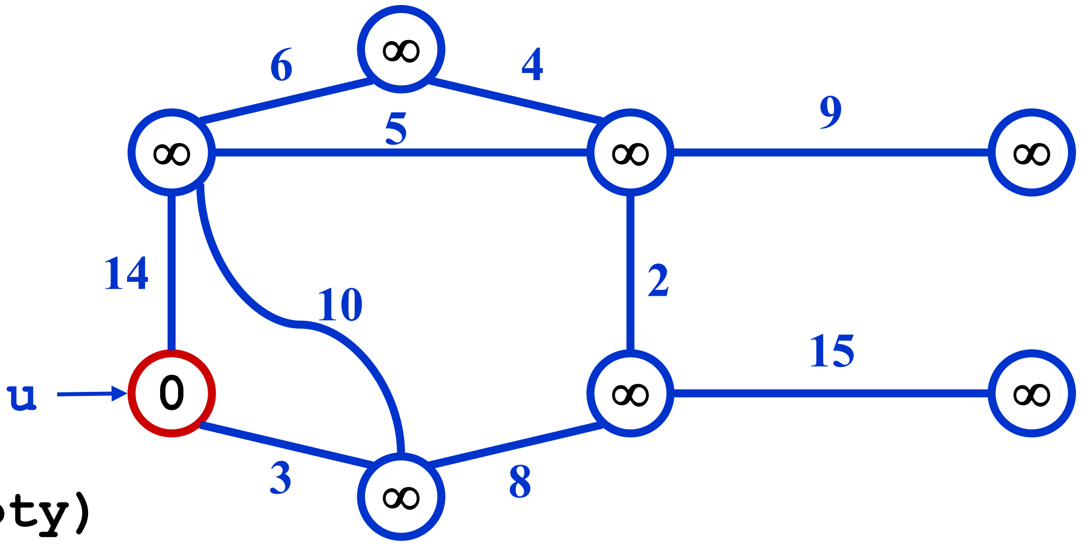
```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



*Pick a start vertex r*

# Prim's Algorithm

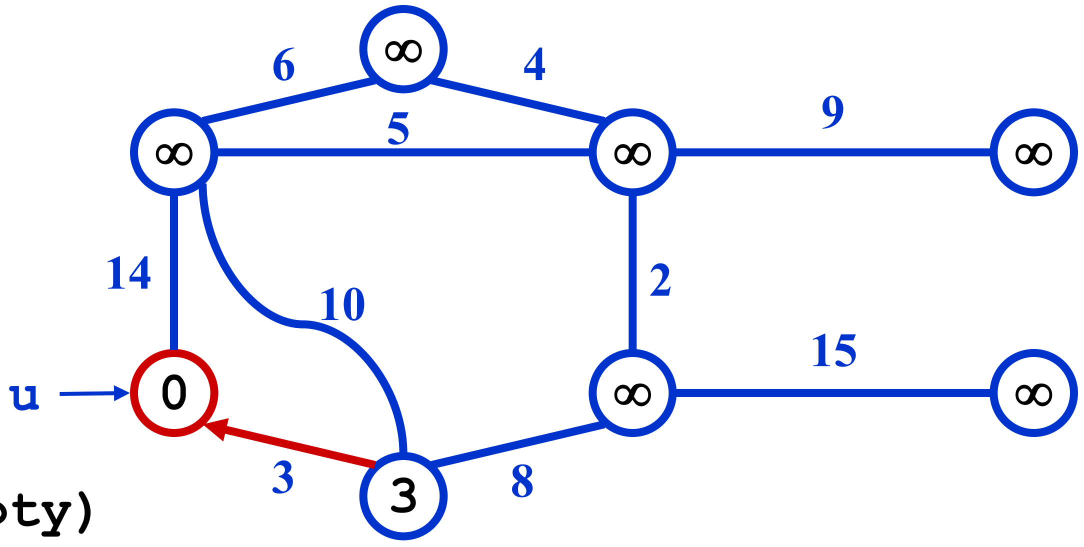
```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



*Red vertices have been removed from Q*

# Prim’s Algorithm

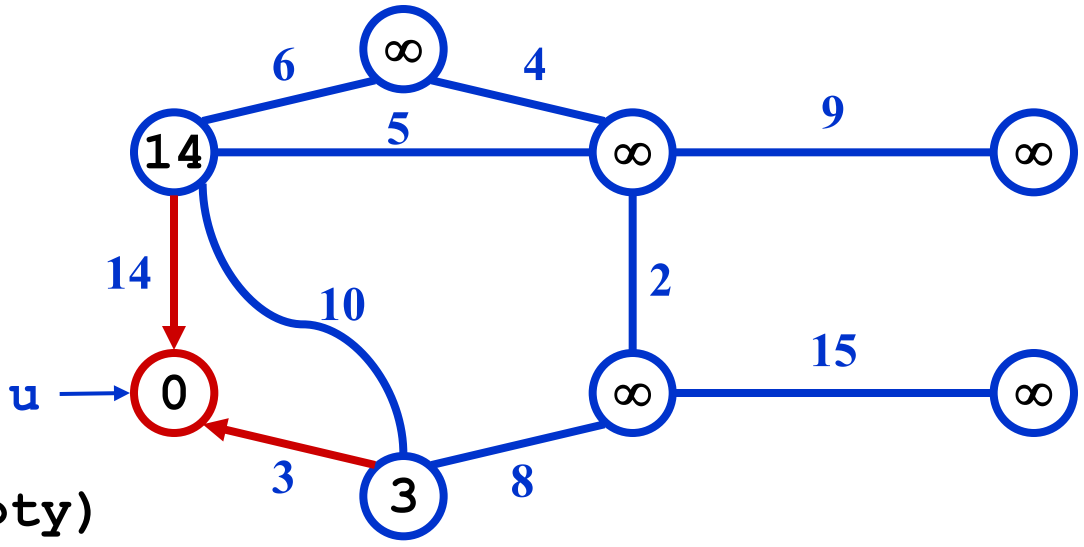
```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



*Red arrows indicate parent pointers*

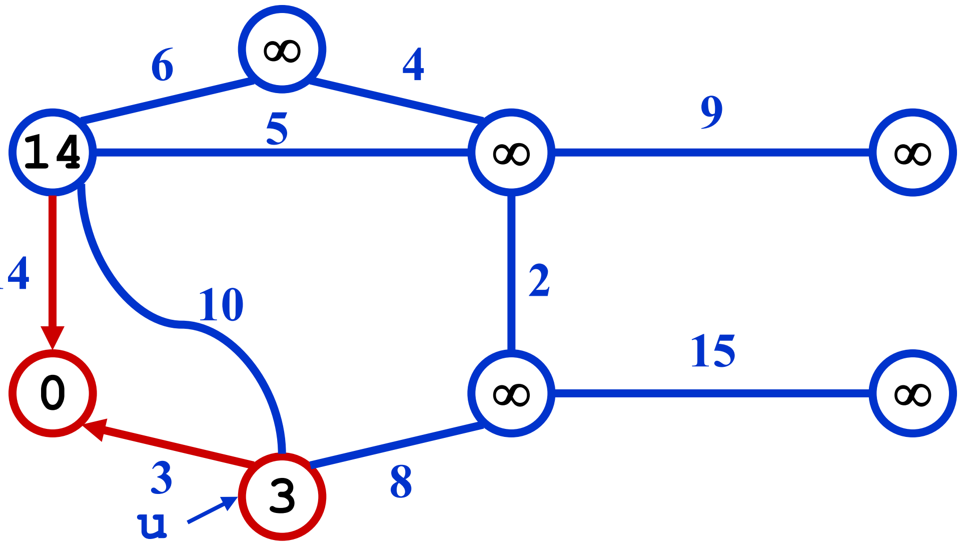
# Prim’s Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



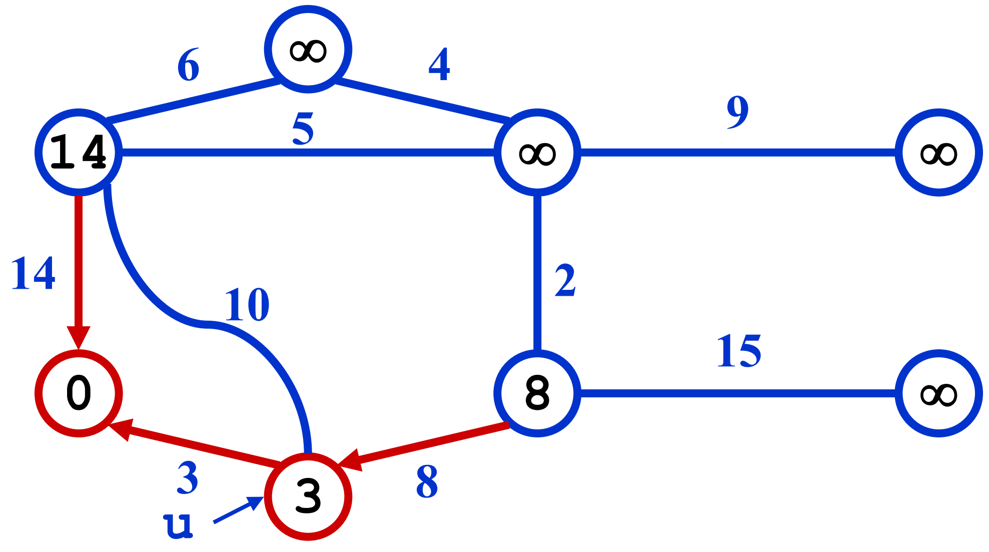
# Prim’s Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



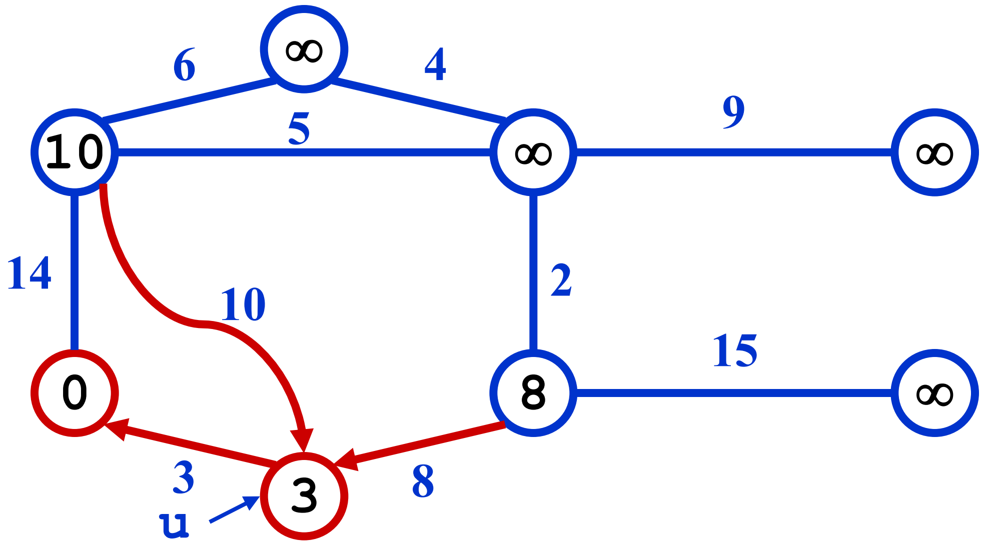
# Prim's Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



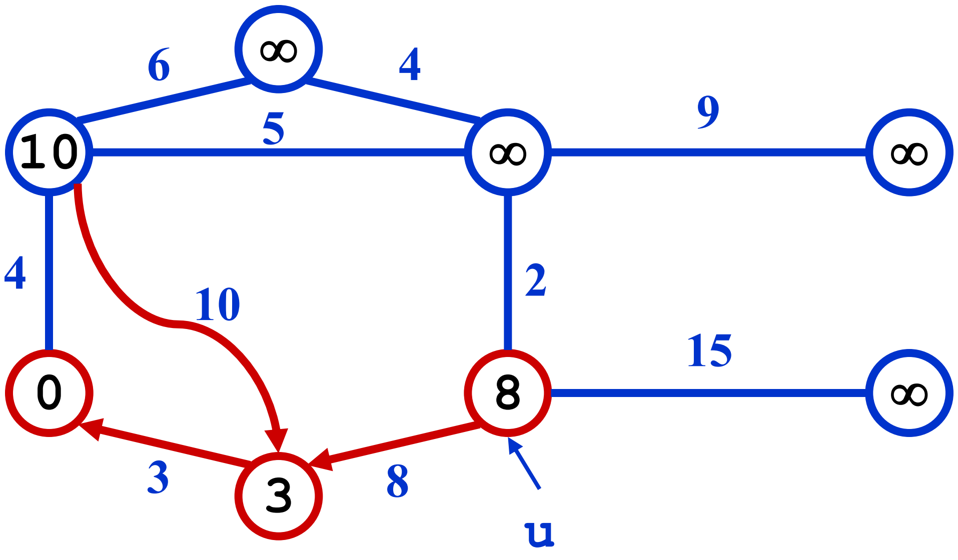
# Prim's Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



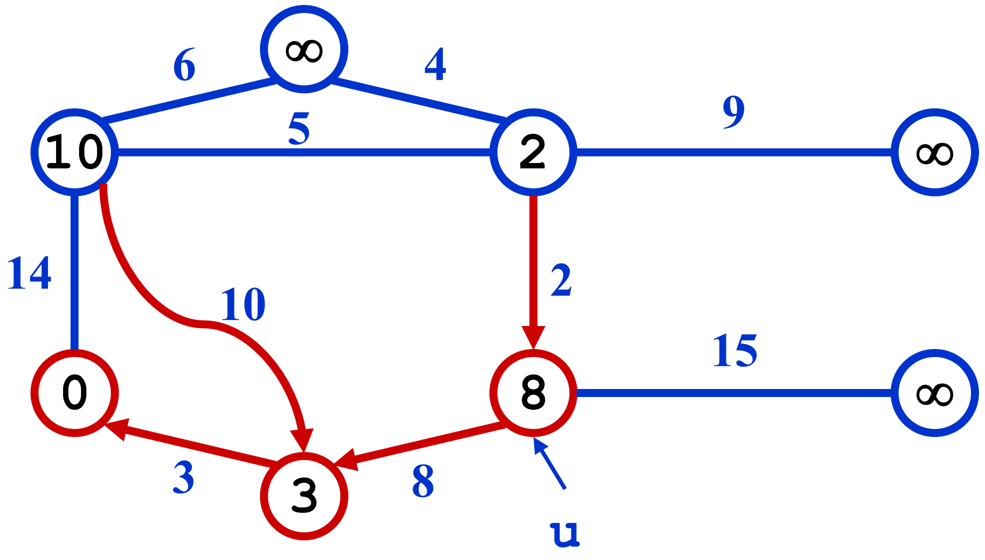
# Prim's Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



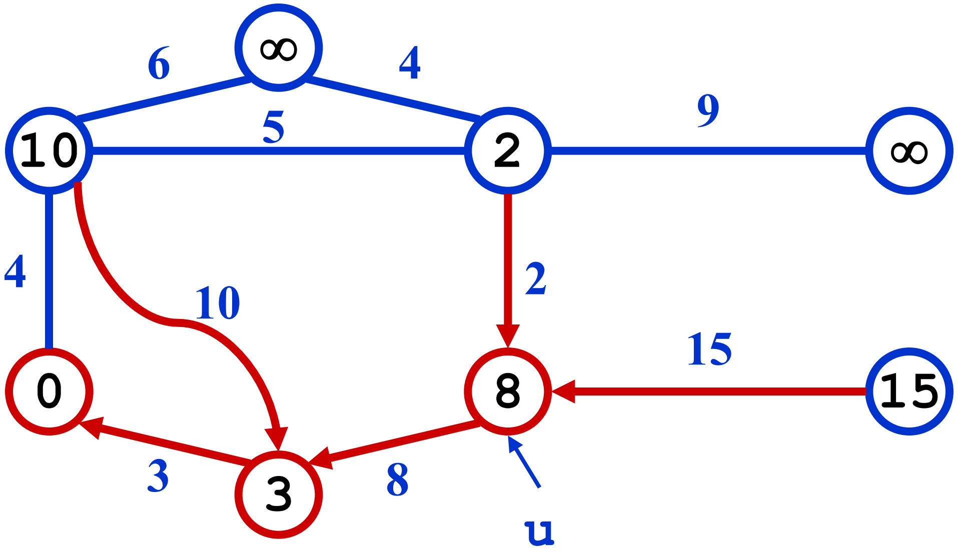
# Prim's Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



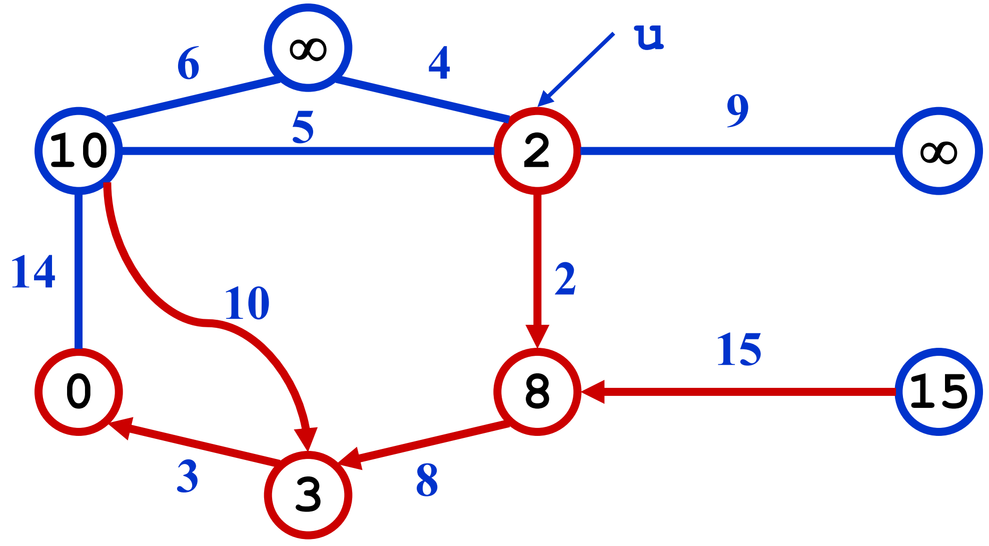
# Prim’s Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



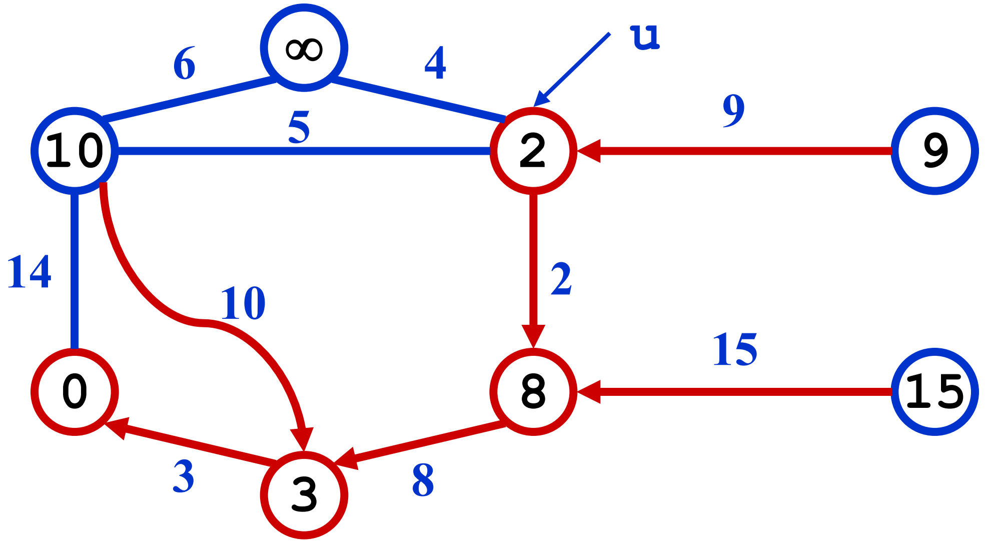
# Prim’s Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



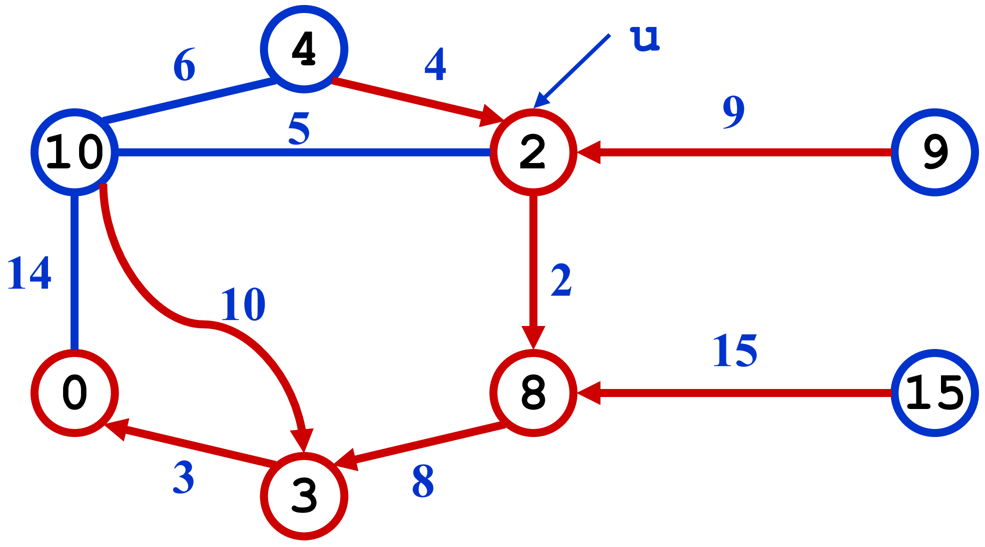
# Prim's Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



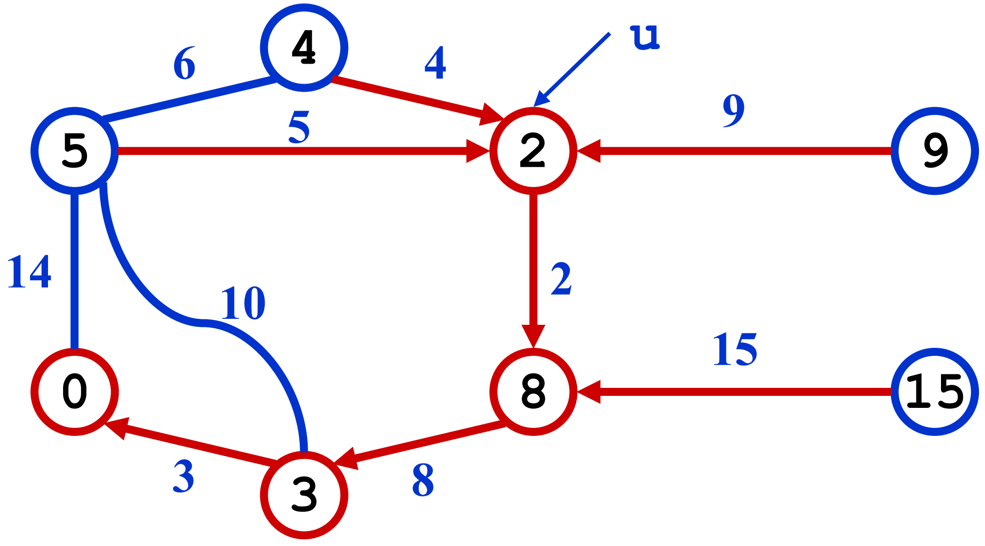
# Prim's Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



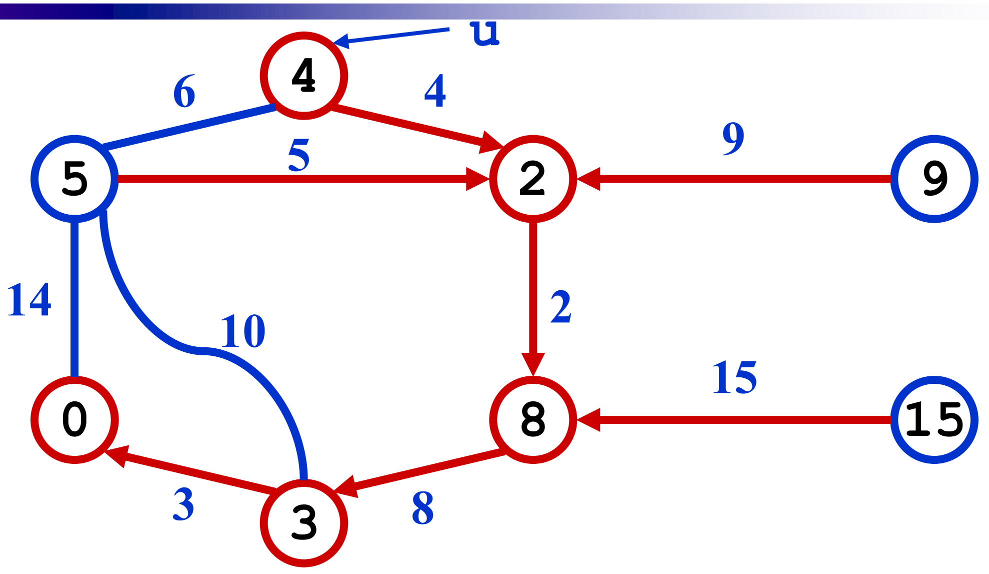
# Prim's Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



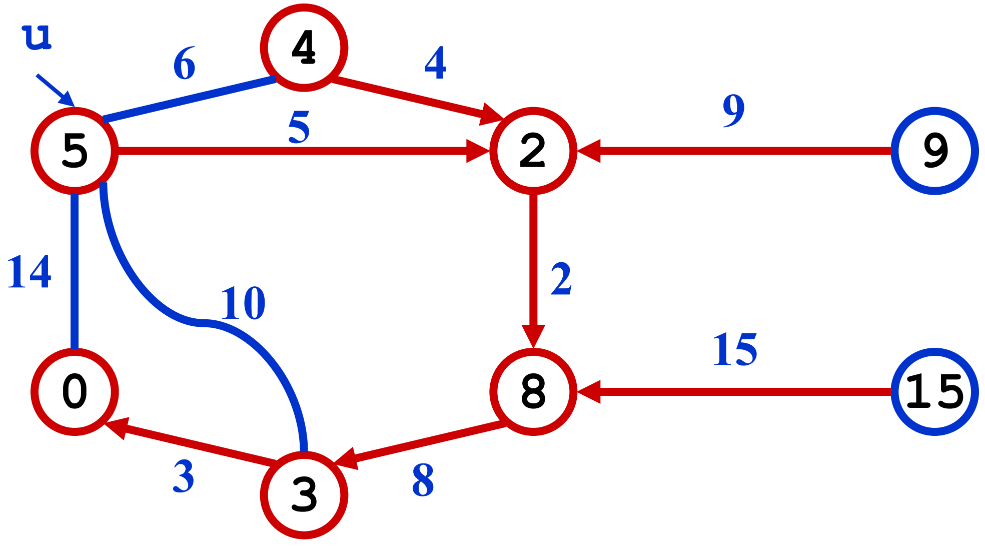
# Prim’s Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



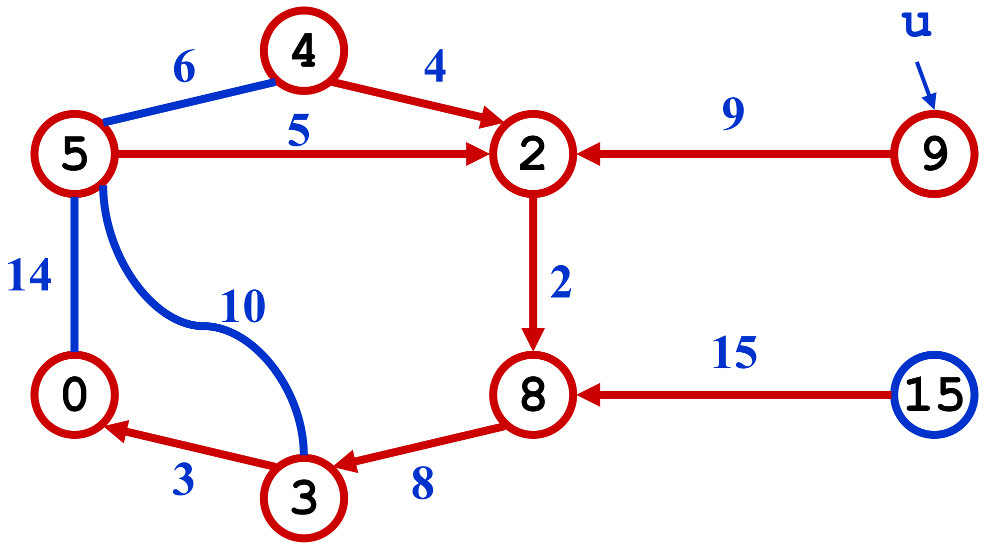
# Prim's Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



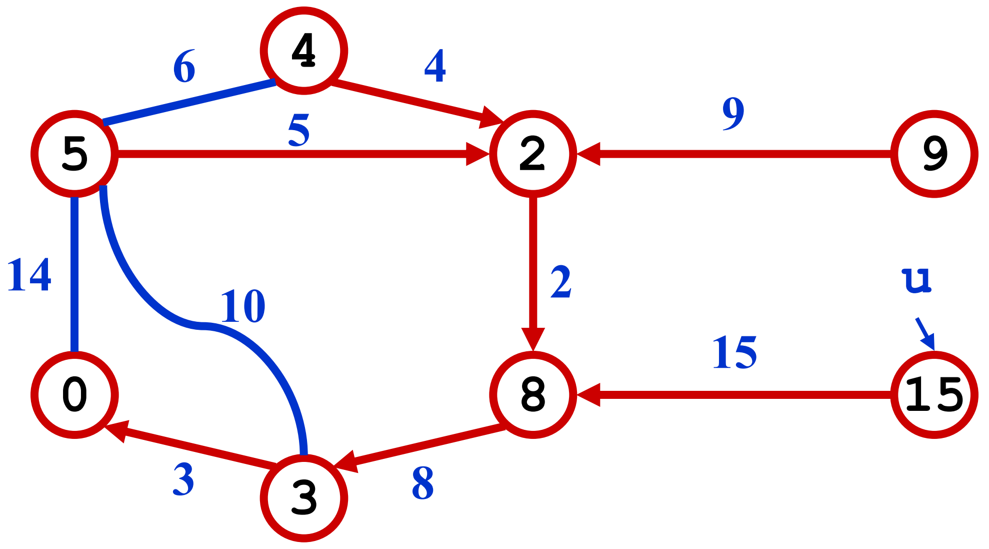
# Prim’s Algorithm

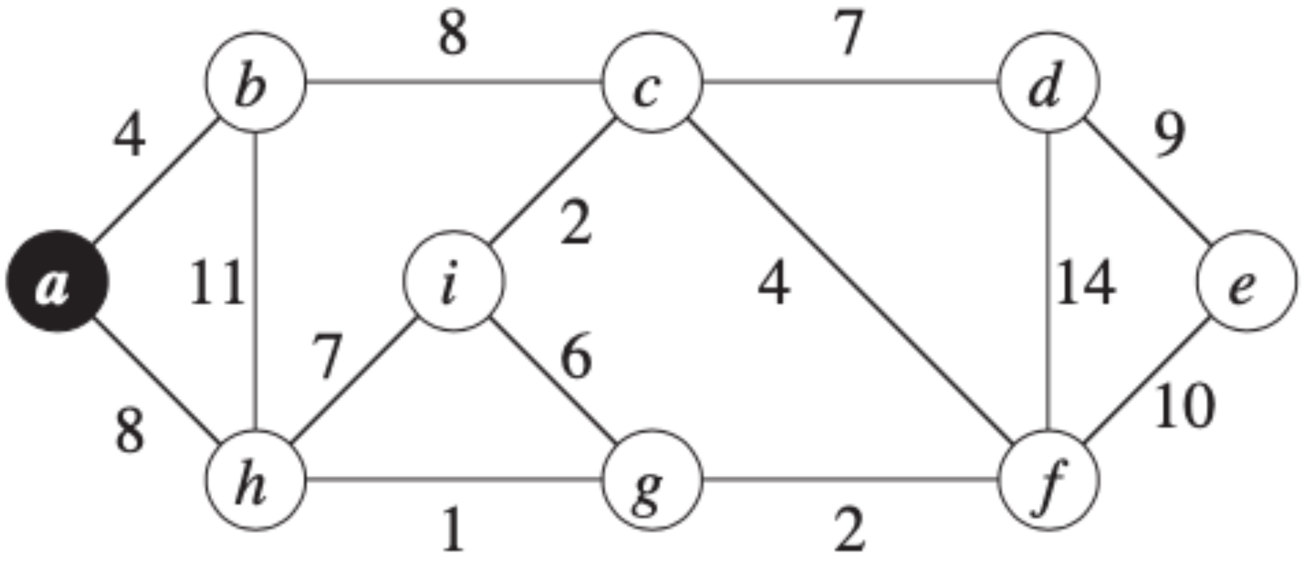
```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



# Prim’s Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```

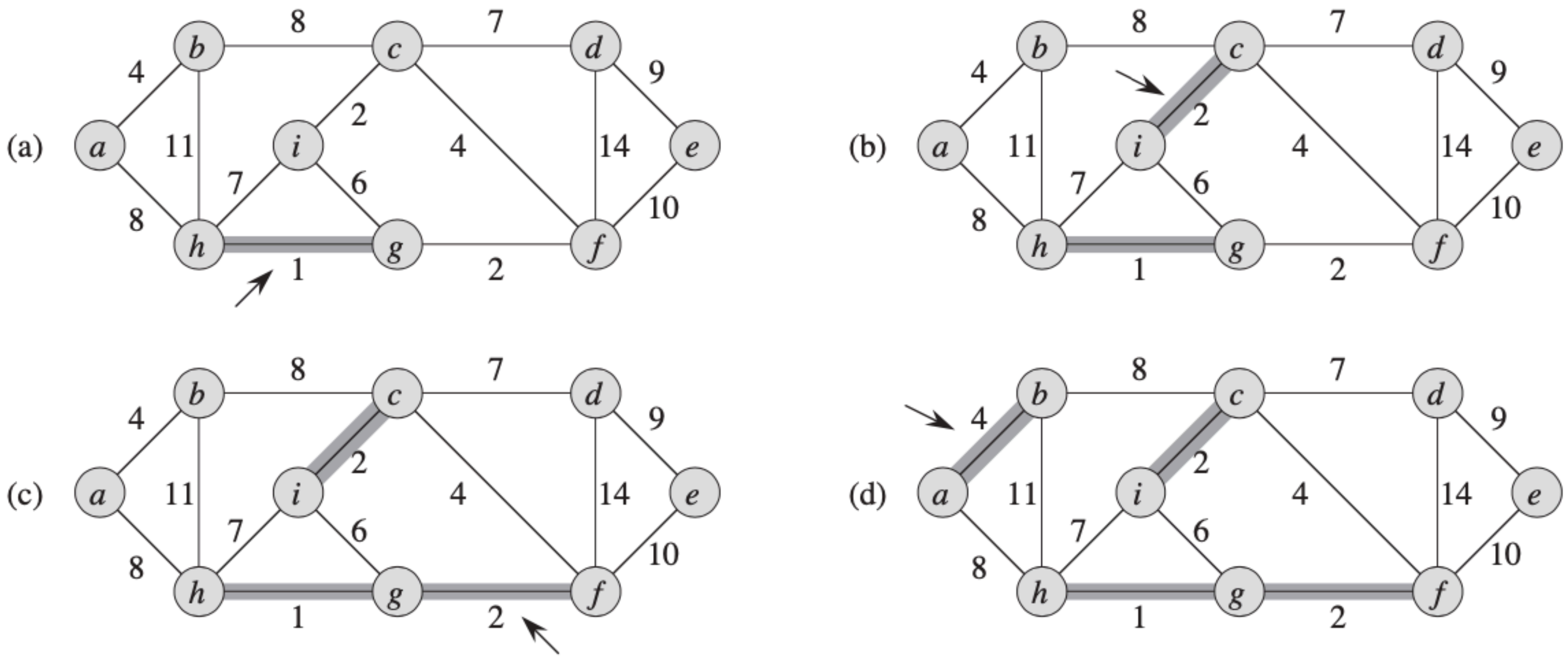


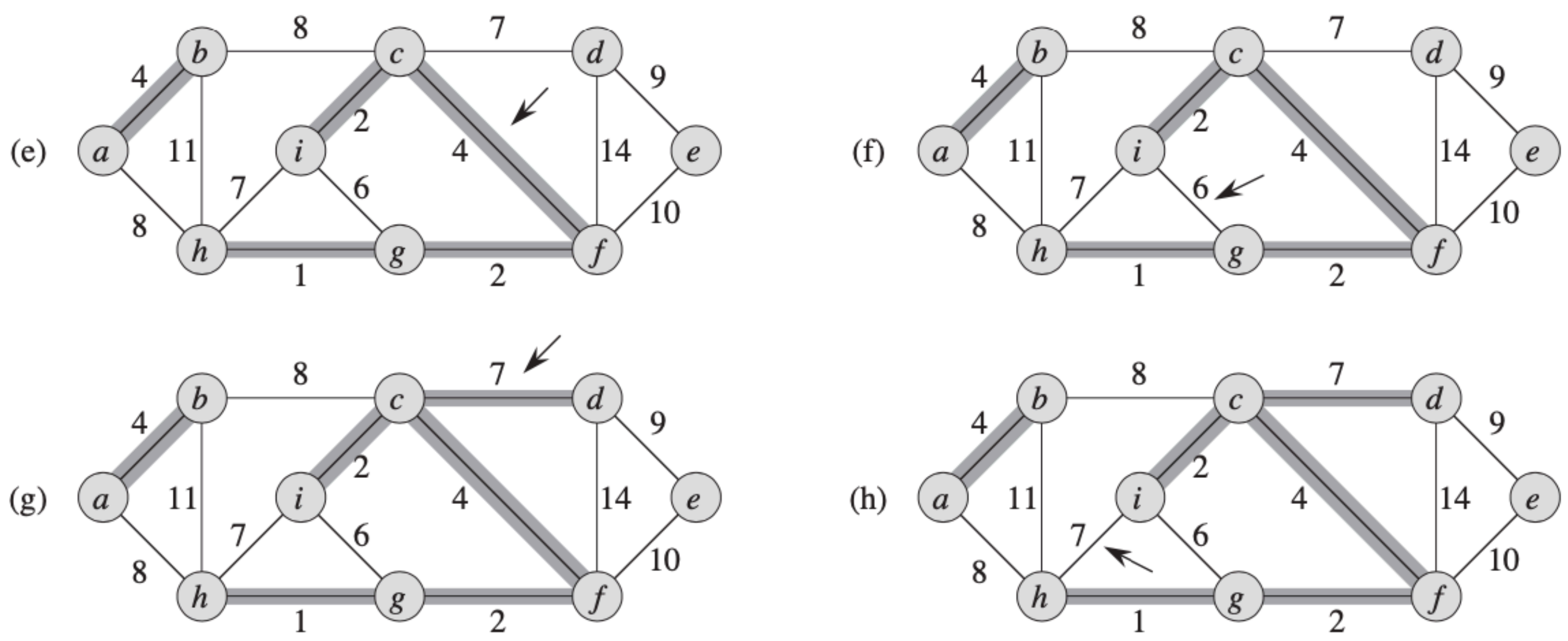

a
b
c
d
e
f
g
h
i
4
8
7
9
11
2
4
14
7
6
10
8
1
2

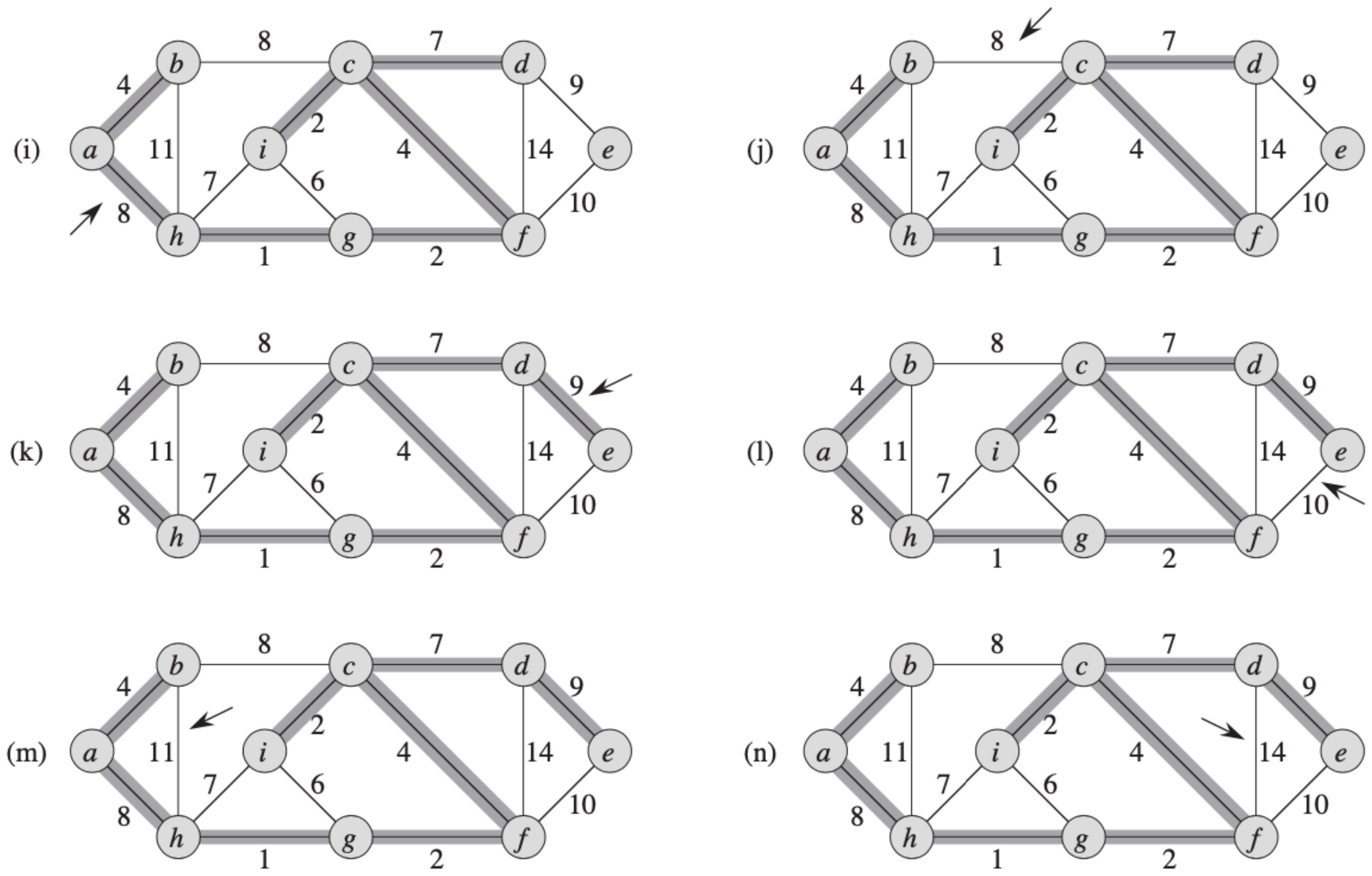
## Kruskal's algorithm

```
MST-KRUSKAL($G, w$)
$A = \emptyset$
for each vertex $v \in G.V$
    MAKE-SET($v$)
sort the edges of $G.E$ into nondecreasing order by weight $w$
for each edge $(u, v) \in G.E$, taken in nondecreasing order by weight
    if FIND-SET($u$) $\neq$ FIND-SET($v$)
        $A = A \cup \{(u, v)\}$
        UNION($u, v$)
return $A$
```

# Kruskal’s example




(e)
(f)
(g)
(h)


(i)
(j)
(k)
(l)
(m)
(n)
a b c d e f g h i
4 8 7 9 11 2 4 14 7 6 8 10 1 2

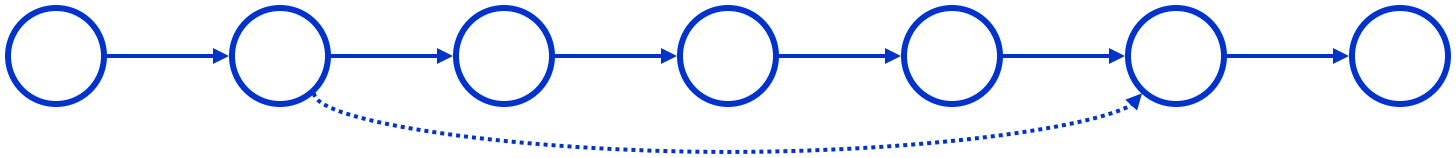
## Single-Source Shortest Path

## Single-Source Shortest Path

- Problem: given a weighted directed graph G, find the minimum-weight path from a given source vertex s to another vertex v
  - “Shortest-path” = minimum weight
  - Weight of path is sum of edges
  - E.g., a road map: what is the shortest path from Louisville to Memphis ?

## Shortest Path Properties

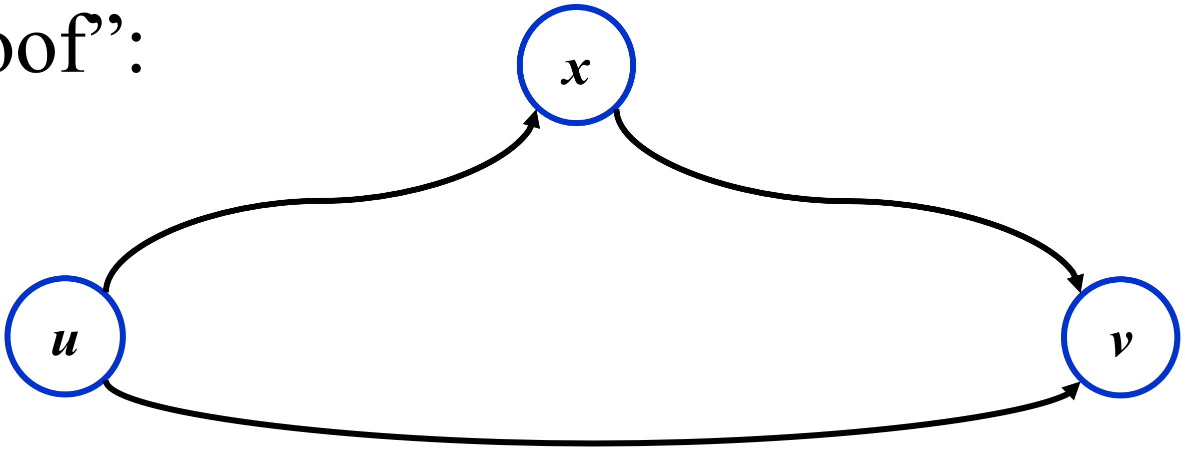
- Again, we have *optimal substructure*: the shortest path consists of shortest subpaths:



  - Proof: suppose some subpath is not a shortest path
    - There must then exist a shorter subpath
    - Could substitute the shorter subpath for a shorter path
    - But then overall path is not shortest path.  Contradiction

## Shortest Path Properties

- Define δ(u,v) to be the weight of the shortest path from u to v
- Shortest paths satisfy the *triangle inequality*: $\delta(u,v) \le \delta(u,x) + \delta(x,v)$
- "Proof":

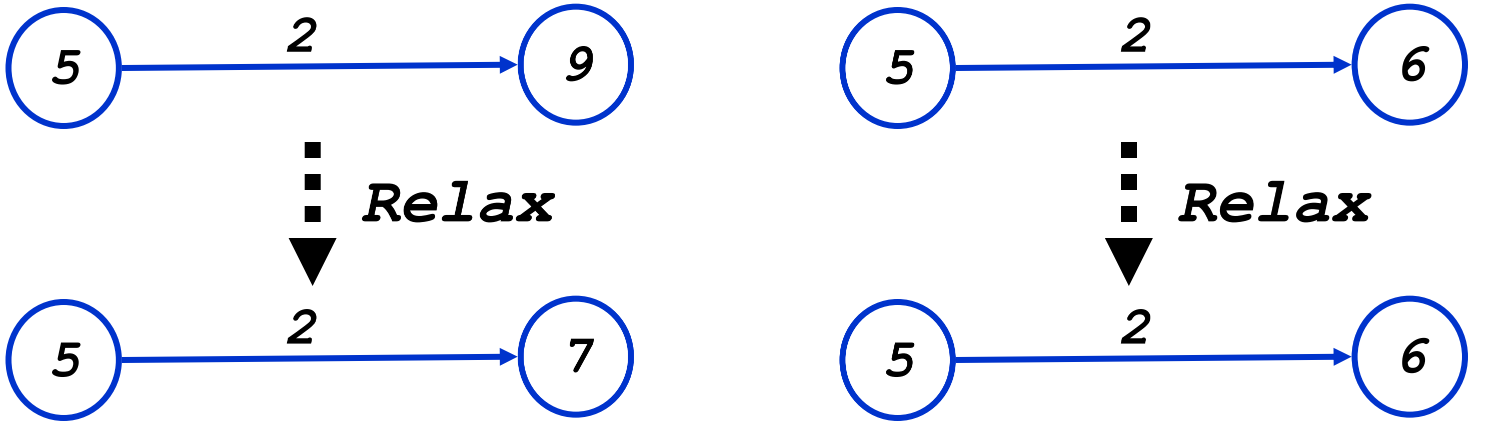


*This path is no longer than any other path*

# Relaxation

- A key technique in shortest path algorithms is *relaxation*
  - Idea: for all *v*, maintain upper bound d[*v*] on δ(s,v)

```
Relax(u,v,w) {
    if (d[v] > d[u]+w) then d[v]=d[u]+w;
}
```



## Bellman-Ford Algorithm

```
BellmanFord()
   for each v ∈ V
      d[v] = ∞;
   d[s] = 0;
   for i=1 to |V|-1
      for each edge (u,v) ∈ E
         Relax(u,v, w(u,v));
   for each edge (u,v) ∈ E
      if (d[v] > d[u] + w(u,v))
           return "no solution";
```

*Initialize d[], which will converge to shortest-path value $\delta$*

*Relaxation: Make |V|-1 passes, relaxing each edge*

*Test for solution*
*Under what condition do we get a solution?*

```
Relax(u,v,w): if (d[v] > d[u]+w) then d[v]=d[u]+w
```

# Bellman-Ford Algorithm

```
BellmanFord()
   for each v ∈ V
      d[v] = ∞;
   d[s] = 0;
   for i=1 to |V|-1
      for each edge (u,v) ∈ E
         Relax(u,v, w(u,v));
   for each edge (u,v) ∈ E
      if (d[v] > d[u] + w(u,v))
           return "no solution";
```

***What will be the running time?***

```
Relax(u,v,w): if (d[v] > d[u]+w) then d[v]=d[u]+w
```

# Bellman-Ford Algorithm

```
BellmanFord()
   for each v ∈ V
      d[v] = ∞;
   d[s] = 0;
   for i=1 to |V|-1
      for each edge (u,v) ∈ E
         Relax(u,v, w(u,v));
   for each edge (u,v) ∈ E
      if (d[v] > d[u] + w(u,v))
           return "no solution";
```

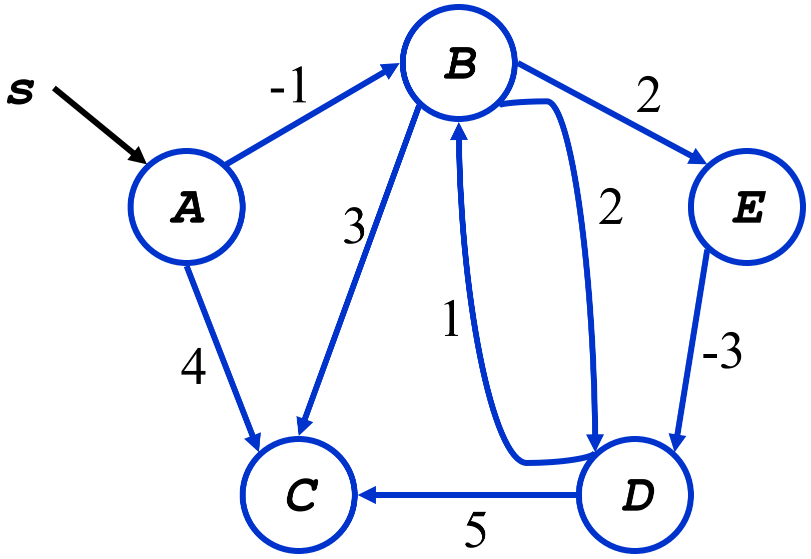
***What will be the running time?***

**A: O(VE)**

```
Relax(u,v,w): if (d[v] > d[u]+w) then d[v]=d[u]+w
```

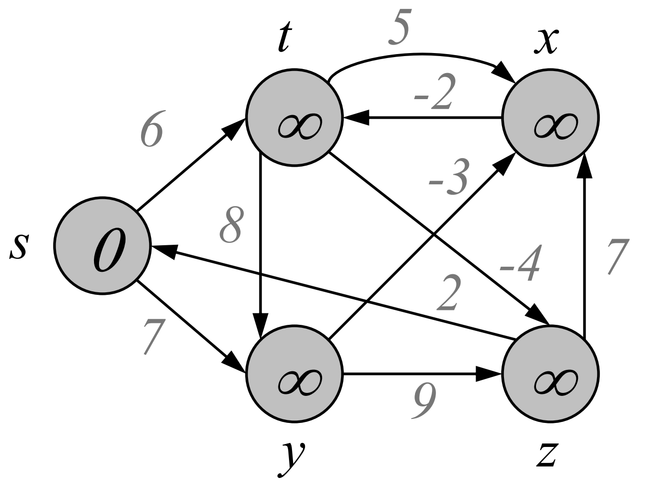
# Bellman-Ford Algorithm

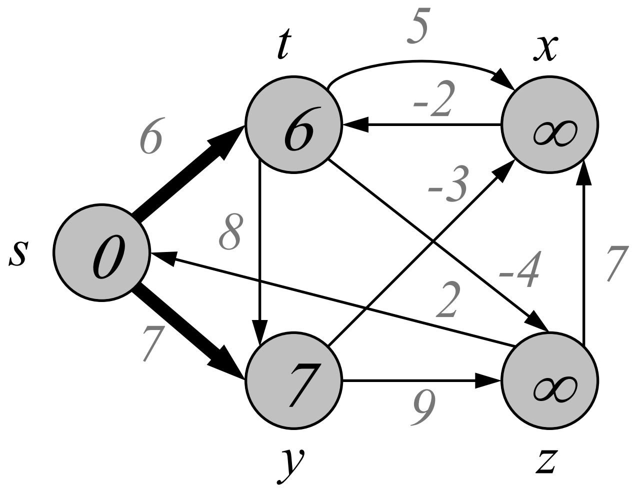
```
BellmanFord()
    for each v ∈ V
        d[v] = ∞;
    d[s] = 0;
    for i=1 to |V|-1
        for each edge (u,v) ∈ E
            Relax(u,v, w(u,v));
    for each edge (u,v) ∈ E
        if (d[v] > d[u] + w(u,v))
            return "no solution";
```

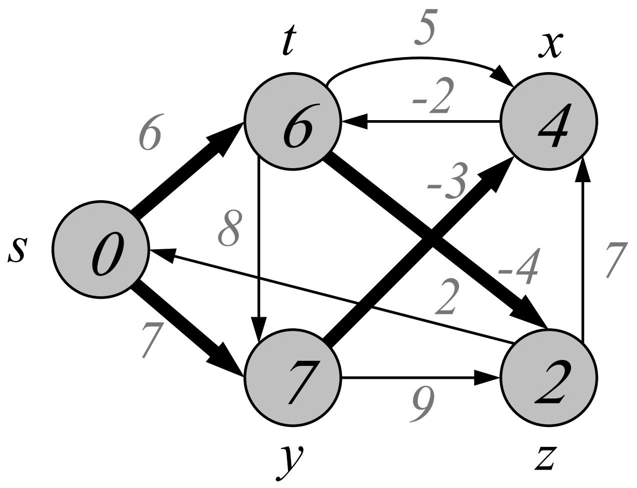


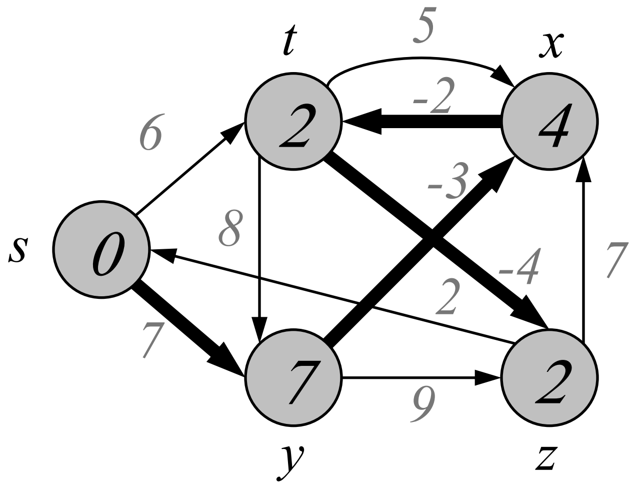
```
Relax(u,v,w): if (d[v] > d[u]+w) then d[v]=d[u]+w
```

# Bellman-Ford Example

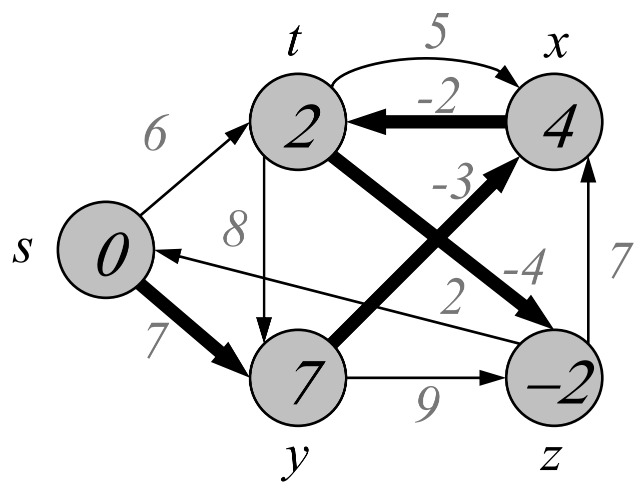






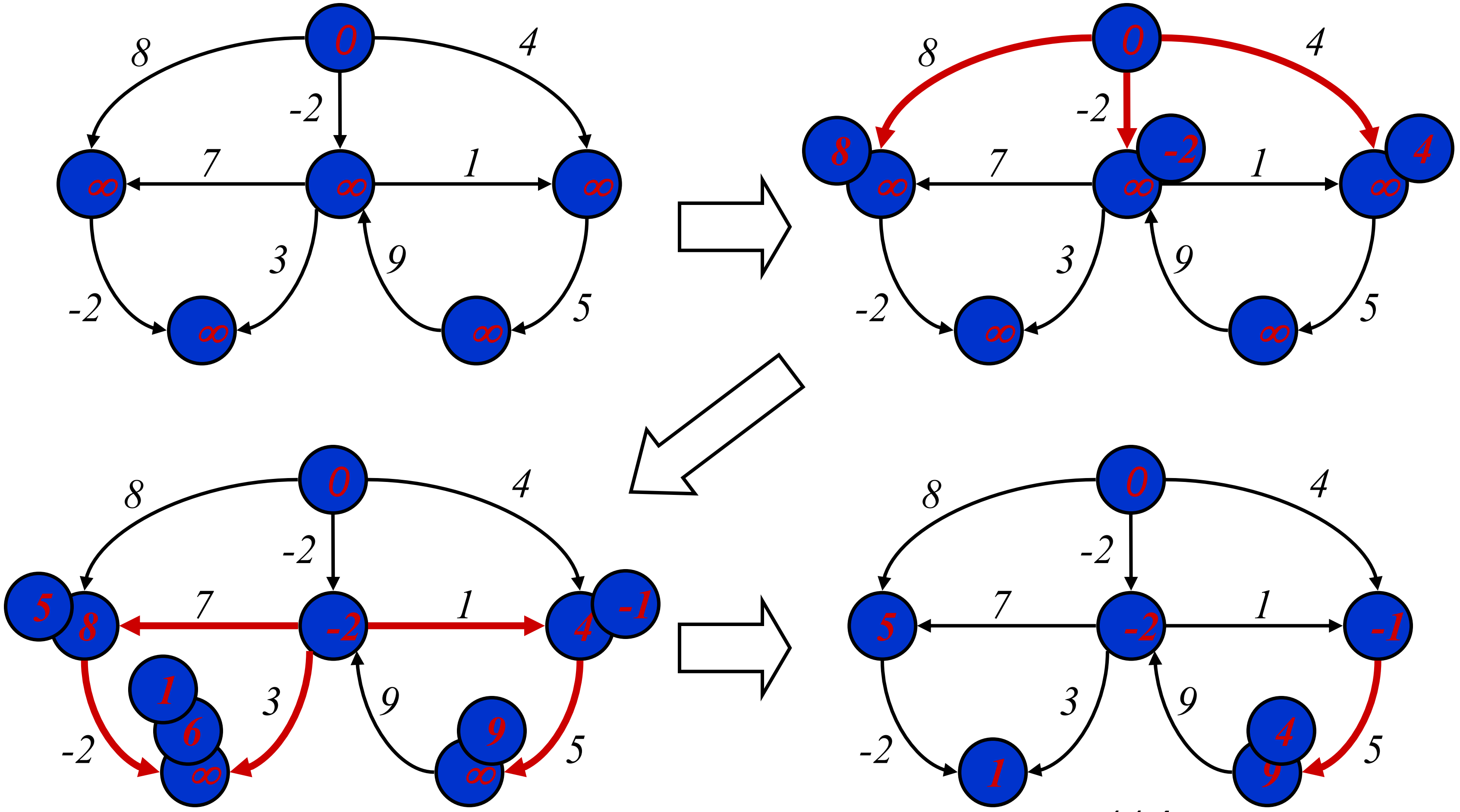


# Bellman-Ford Example



- Bellman-Ford running time:
  - $(|V|-1)|E| + |E| = \Theta(VE)$

# Bellman-Ford Example



# Dijkstra's algorithm

**Dijkstra's algorithm** - is a solution to the single-source shortest path problem in graph theory.

Works on both directed and undirected graphs. However, all edges must have nonnegative weights.

Approach: Greedy

Input: Weighted graph G={E,V} and source vertex $v \in V$, such that all edge weights are nonnegative

Output: Lengths of shortest paths (or the shortest paths themselves) from a given source vertex $v \in V$ to all other vertices

# Dijkstra's algorithm - Pseudocode

```
$dist[s] \leftarrow 0$                                  (distance to source vertex is zero)
for all $v \in V - \{s\}$
        do $dist[v] \leftarrow \infty$                  (set all other distances to infinity)
$S \leftarrow \emptyset$                                (S, the set of visited vertices is initially empty)
$Q \leftarrow V$                              (Q, the queue initially contains all vertices)
while $Q \neq \emptyset$                                (while the queue is not empty)
do  $u \leftarrow$ mindistance$(Q, dist)$               (select the element of Q with the min. distance)
    $S \leftarrow S \cup \{u\}$                         (add u to list of visited vertices)
    for all $v \in neighbors[u]$
        do if  $dist[v] > dist[u] + w(u, v)$                  (if new shortest path found)
               then    $d[v] \leftarrow d[u] + w(u, v)$    (set new value of shortest path)
                    (if desired, add traceback code)
return $dist$
```

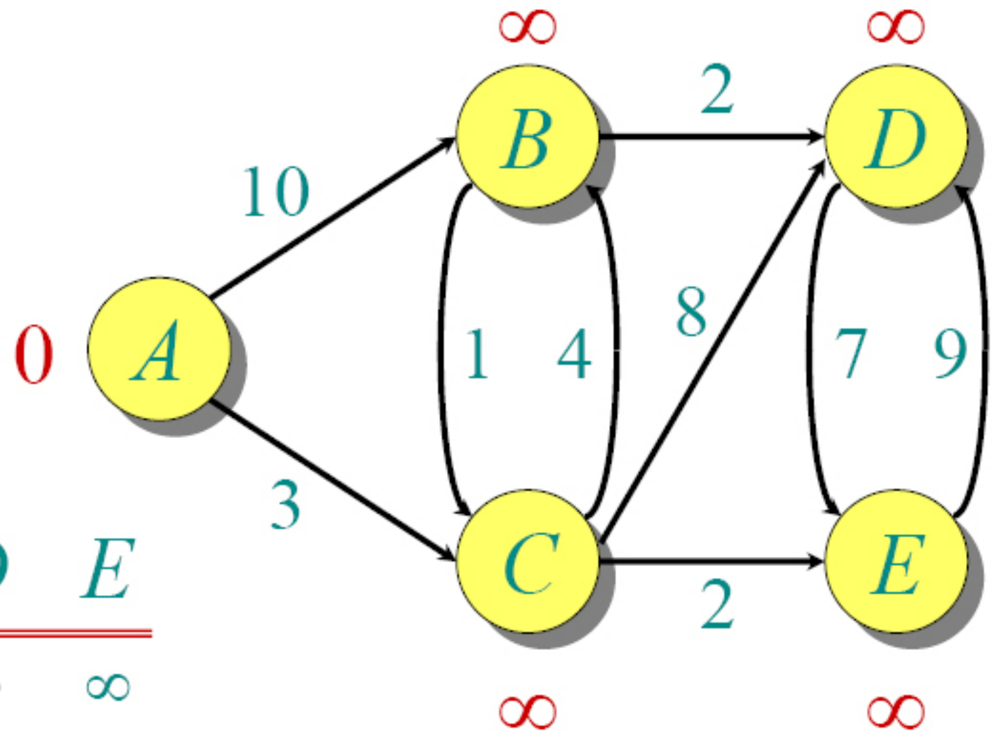
## Dijkstra’s Algorithm

- If no negative edge weights, we can beat BF
- Similar to breadth-first search
  - Grow a tree gradually, advancing from vertices taken from a queue
- Also similar to Prim’s algorithm for MST
  - Use a priority queue keyed on d[v]

# Dijkstra Animated Example

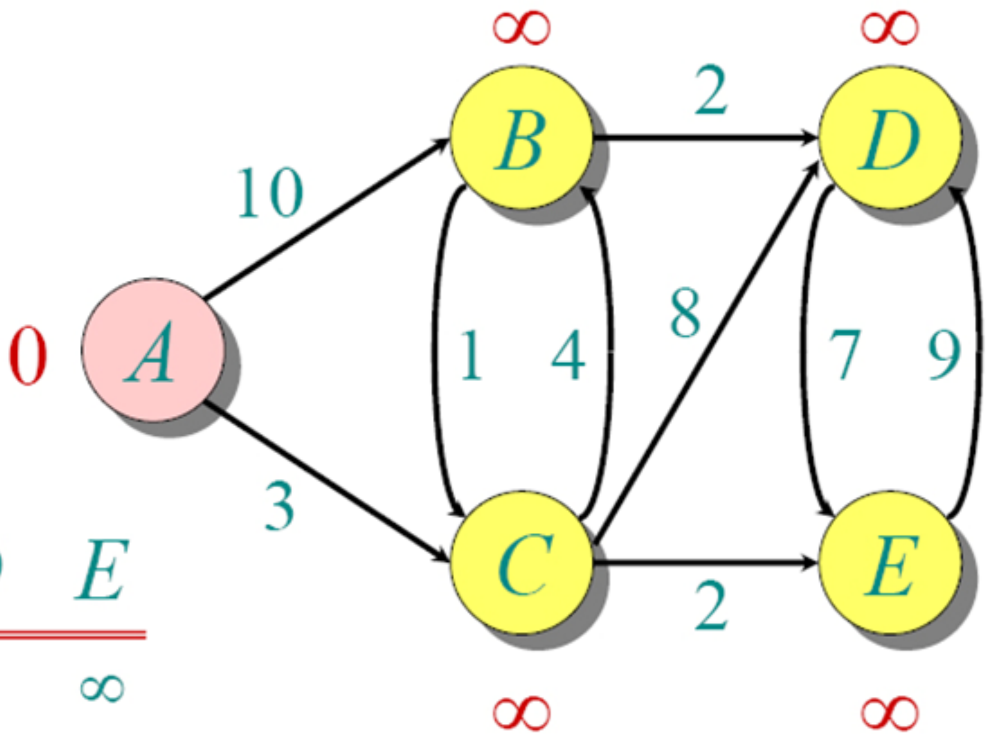
**Initialize:**

| Q: | A | B | C | D | E |
|---|---|---|---|---|---|
| | 0 | ∞ | ∞ | ∞ | ∞ |



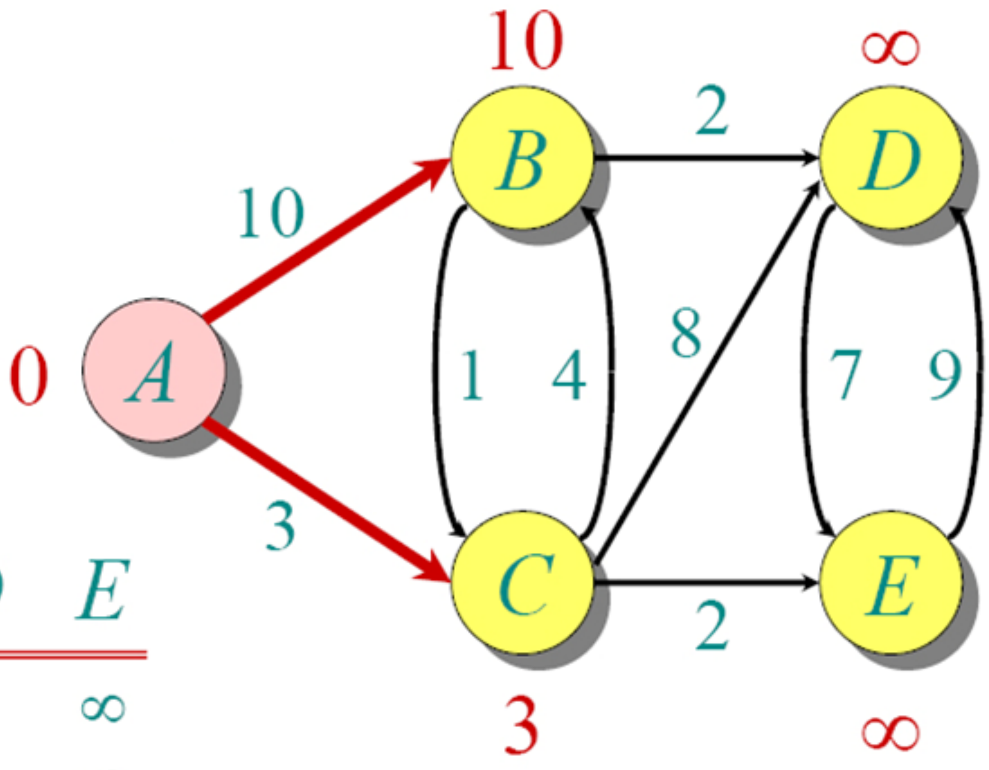
$S$: {}

# Dijkstra Animated Example



| Q: | A | B | C | D | E |
|---|---|---|---|---|---|
| | 0 | ∞ | ∞ | ∞ | ∞ |

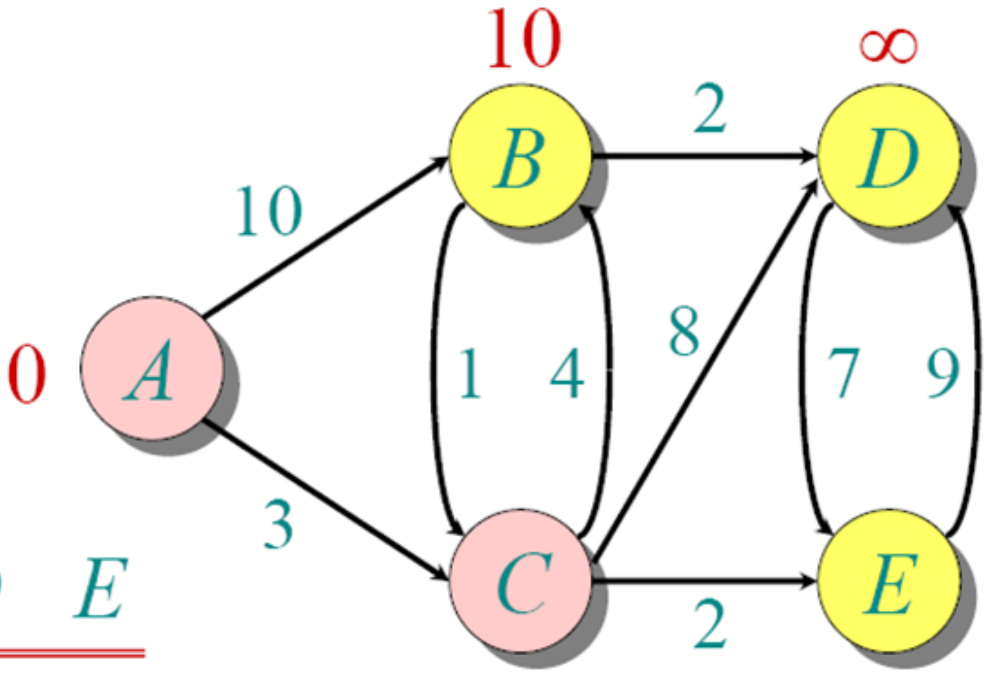
# Dijkstra Animated Example



| Q: | A | B | C | D | E |
|---|---|---|---|---|---|
| | 0 | ∞ | ∞ | ∞ | ∞ |
| | | 10 | 3 | ∞ | ∞ |

S: { A }

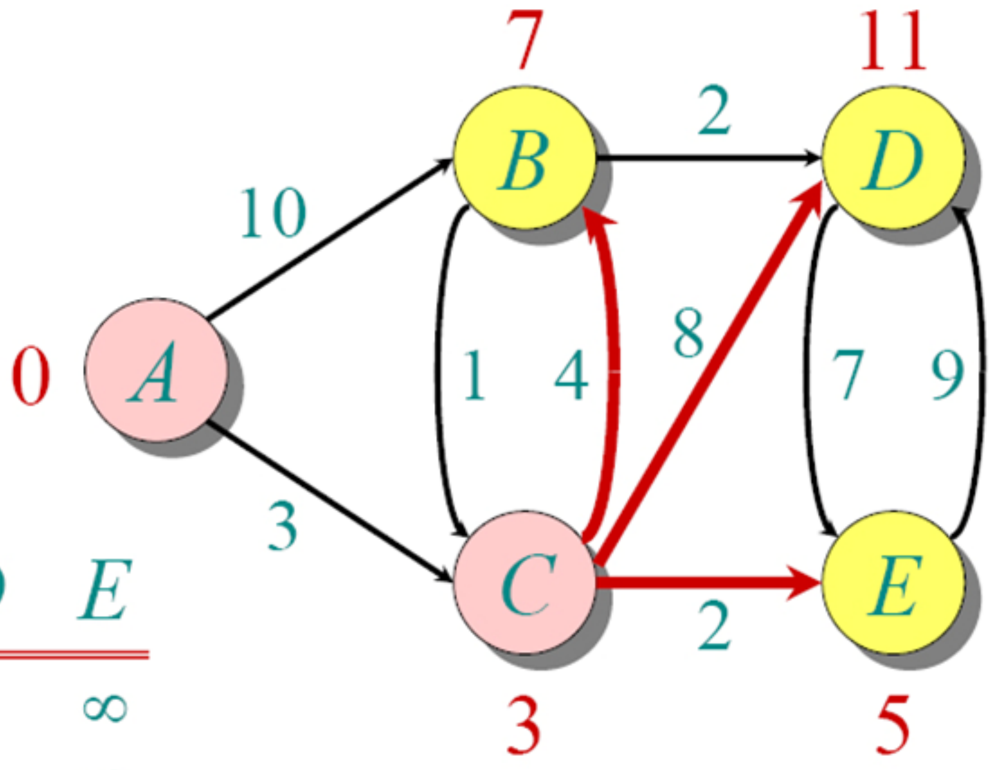
# Dijkstra Animated Example



| *Q:* | *A* | *B* | *C* | *D* | *E* |
|---|---|---|---|---|---|
| | 0 | ∞ | ∞ | ∞ | ∞ |
| | | 10 | 3 | ∞ | ∞ |

*S*: { *A*, *C* }

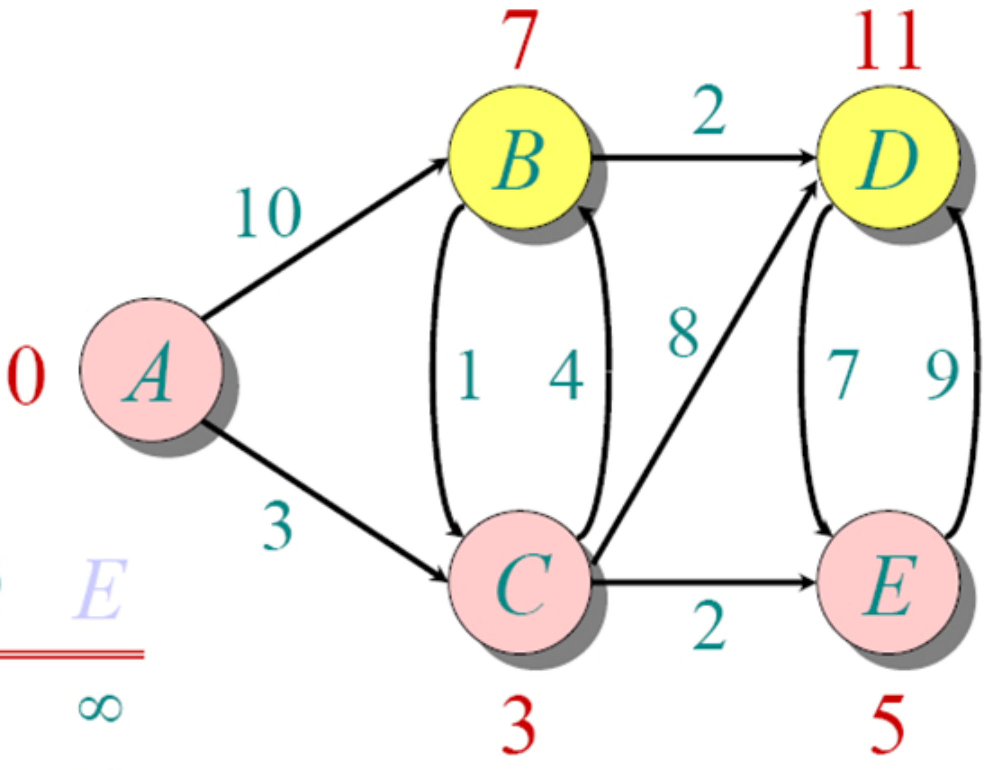
# Dijkstra Animated Example



| Q: | A | B | C | D | E |
|---|---|---|---|---|---|
| | 0 | ∞ | ∞ | ∞ | ∞ |
| | | 10 | 3 | ∞ | ∞ |
| | | 7 | | 11 | 5 |

*S*: { *A*, *C* }

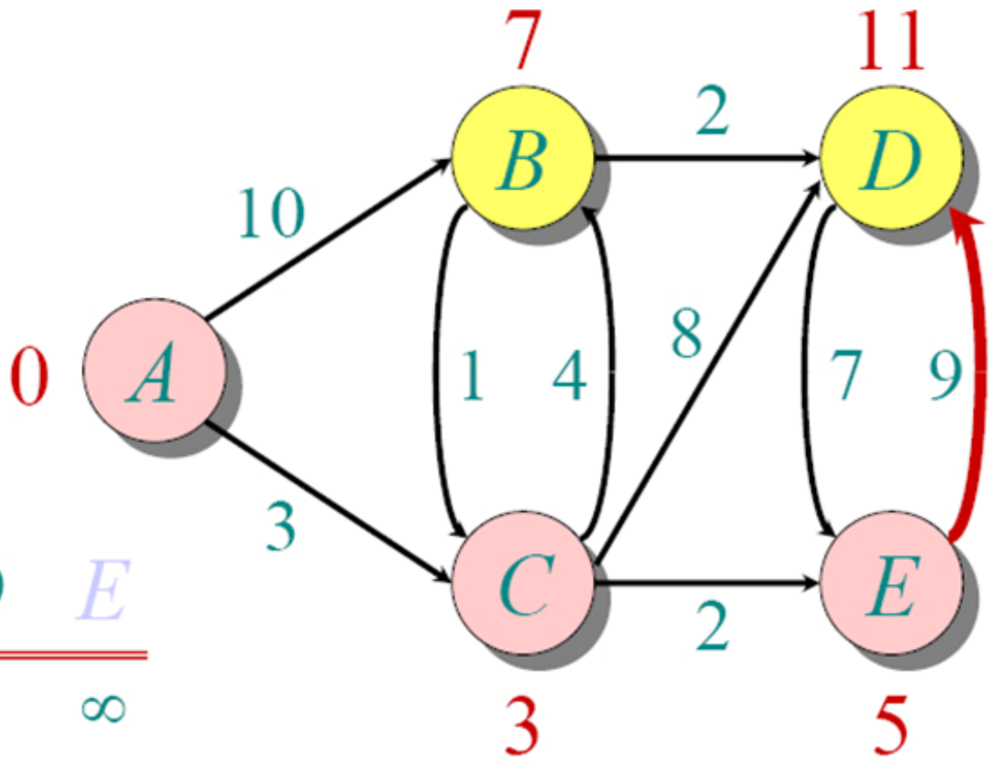
# Dijkstra Animated Example



| Q: | A | B | C | D | E |
|---|---|---|---|---|---|
| | 0 | ∞ | ∞ | ∞ | ∞ |
| | | 10 | 3 | ∞ | ∞ |
| | | 7 | | 11 | 5 |

S: { A, C, E }

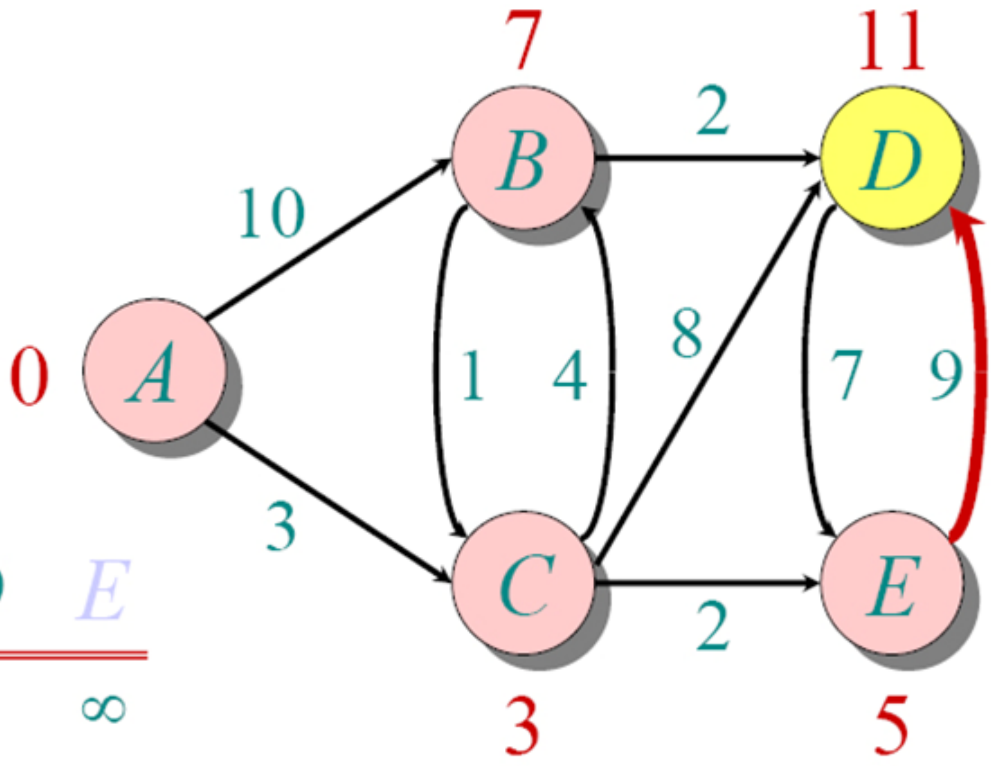
# Dijkstra Animated Example



| Q: | A | B | C | D | E |
|---|---|---|---|---|---|
| | 0 | ∞ | ∞ | ∞ | ∞ |
| | | 10 | 3 | ∞ | ∞ |
| | | 7 | | 11 | 5 |
| | | 7 | | 11 | |

*S*: { *A*, *C*, *E* }

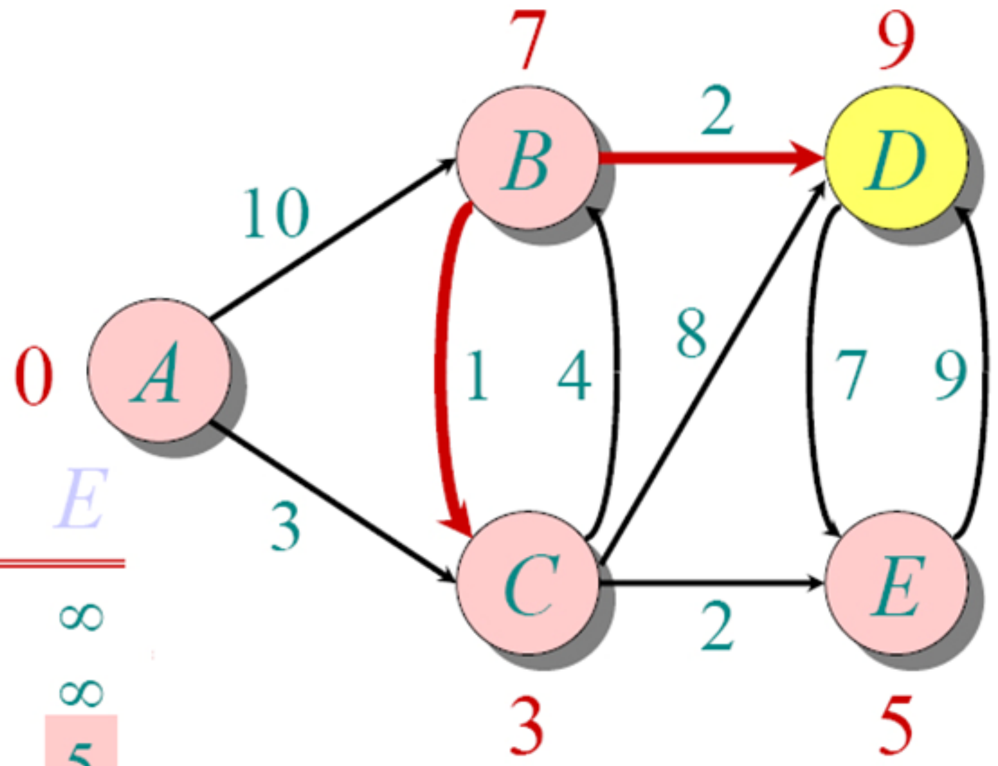
# Dijkstra Animated Example



| Q: | A | B | C | D | E |
|---|---|---|---|---|---|
| | 0 | ∞ | ∞ | ∞ | ∞ |
| | | 10 | 3 | ∞ | ∞ |
| | | 7 | | 11 | 5 |
| | | 7 | | 11 | |

S: { A, C, E, B }

# Dijkstra Animated Example

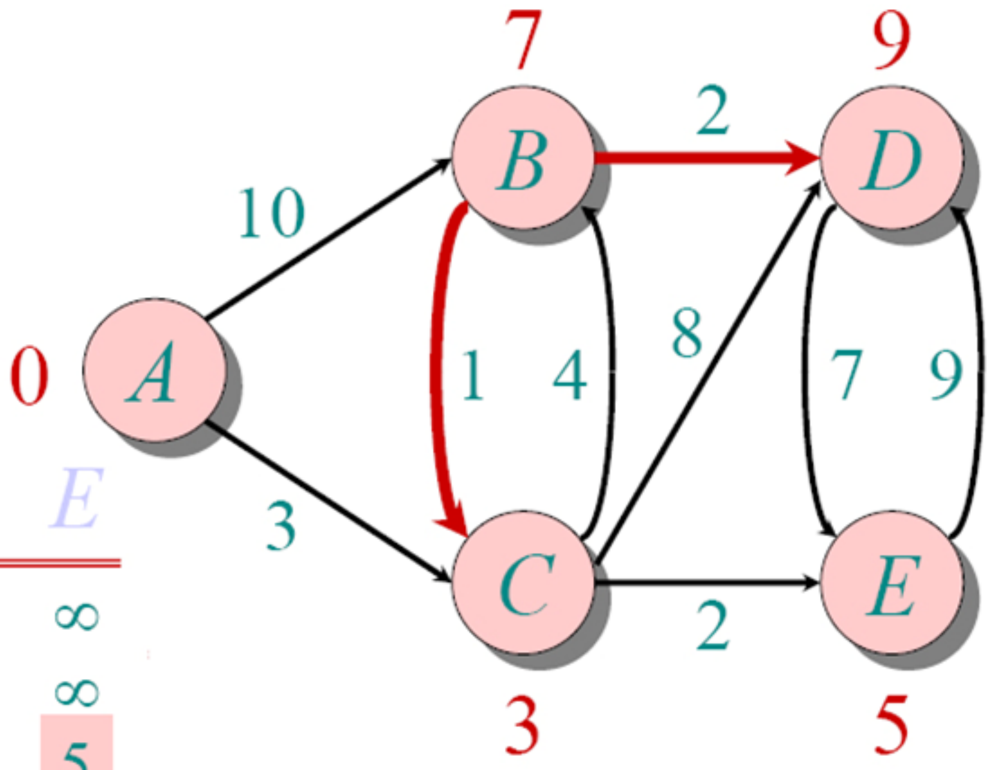


Q:

| A | B | C | D | E |
|---|---|---|---|---|
| 0 | ∞ | ∞ | ∞ | ∞ |
|  | 10 | 3 | ∞ | ∞ |
|  | 7 |  | 11 | 5 |
|  | 7 |  | 11 |  |
|  |  |  | 9 |  |

S: { A, C, E, B }

# Dijkstra Animated Example



| Q: | *A* | *B* | *C* | *D* | *E* |
|---|---|---|---|---|---|
| | 0 | ∞ | ∞ | ∞ | ∞ |
| | | 10 | 3 | ∞ | ∞ |
| | | 7 | | 11 | 5 |
| | | 7 | | 11 | |
| | | | | 9 | |

*S*: { *A*, *C*, *E*, *B*, *D* }

## Implementations and Running Times

For sparse graphs, or graphs with very few edges and many nodes, it can be implemented more efficiently storing the graph in an adjacency list using a binary heap or priority queue. This will produce a running time of

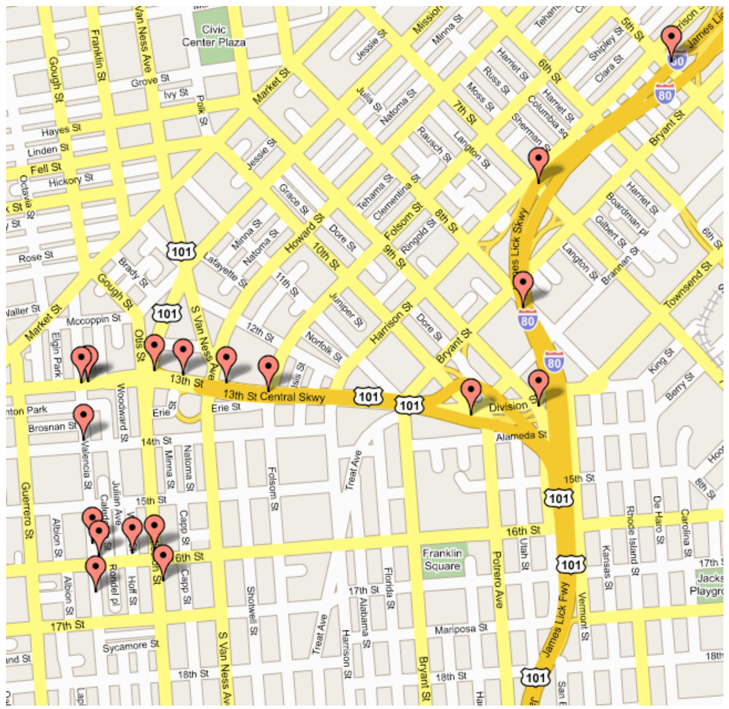
$O((|E|+|V|) \log |V|)$

## Dijkstra's Algorithm - Why use it?

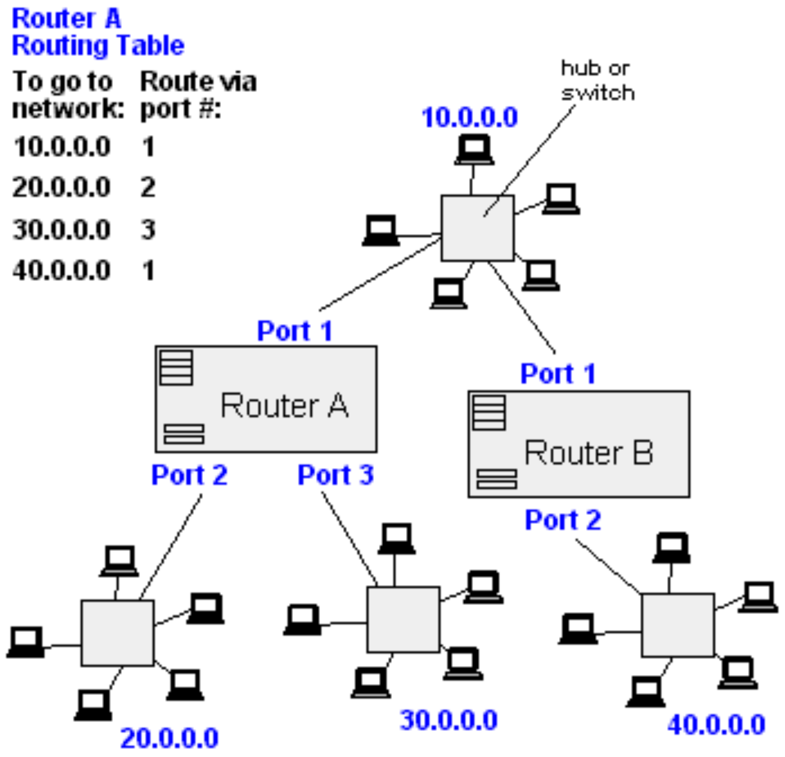
- As mentioned, Dijkstra’s algorithm calculates the shortest path to every vertex.
- However, it is about as computationally expensive to calculate the shortest path from vertex *u* to every vertex using Dijkstra’s as it is to calculate the shortest path to some particular vertex *v*.
- Therefore, anytime we want to know the optimal path to some other vertex from a determined origin, we can use Dijkstra’s algorithm.

# Applications of Dijkstra's Algorithm

- Traffic Information Systems are most prominent use
- Mapping (Map Quest, Google Maps)
- Routing Systems



From Computer Desktop Encyclopedia
© 1998 The Computer Language Co. Inc.

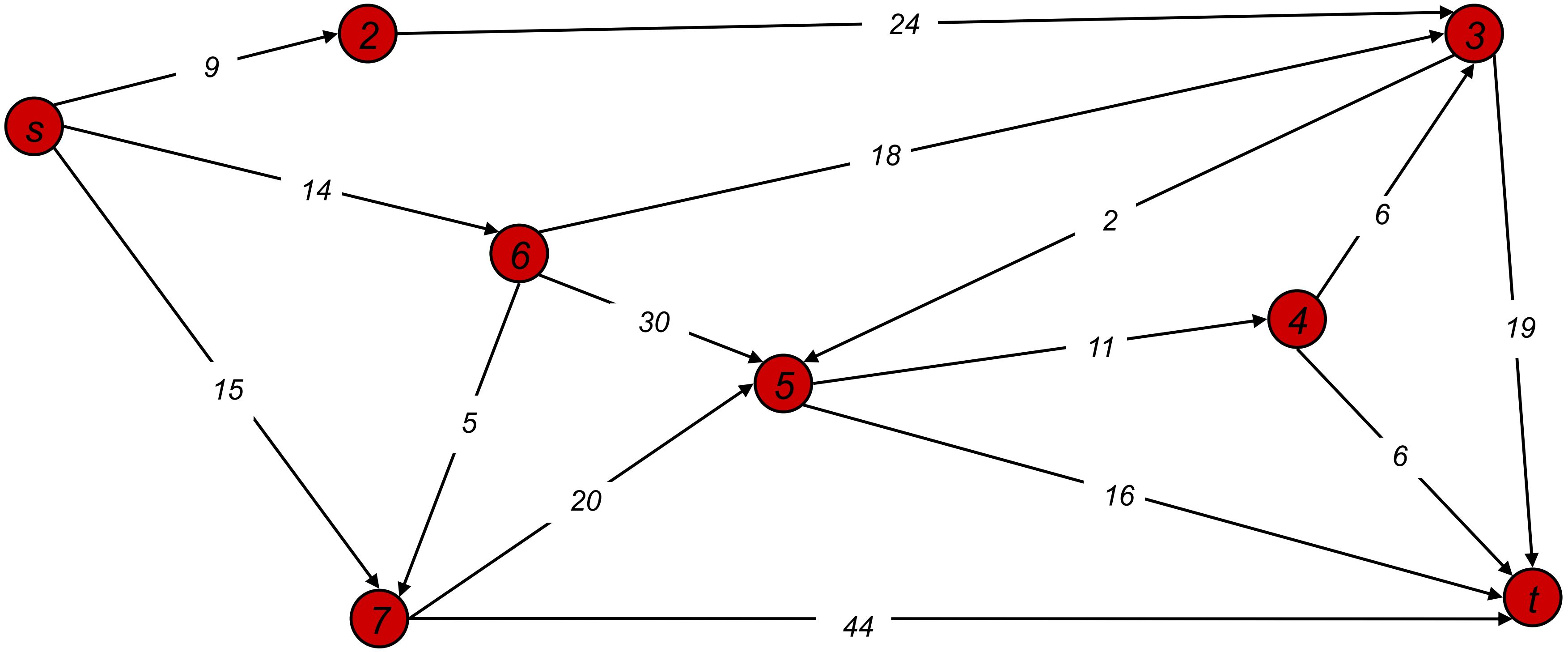


Router A
Routing Table

| To go to network: | Route via port #: |
|---|---|
| 10.0.0.0 | 1 |
| 20.0.0.0 | 2 |
| 30.0.0.0 | 3 |
| 40.0.0.0 | 1 |

# Dijkstra's Shortest Path Algorithm

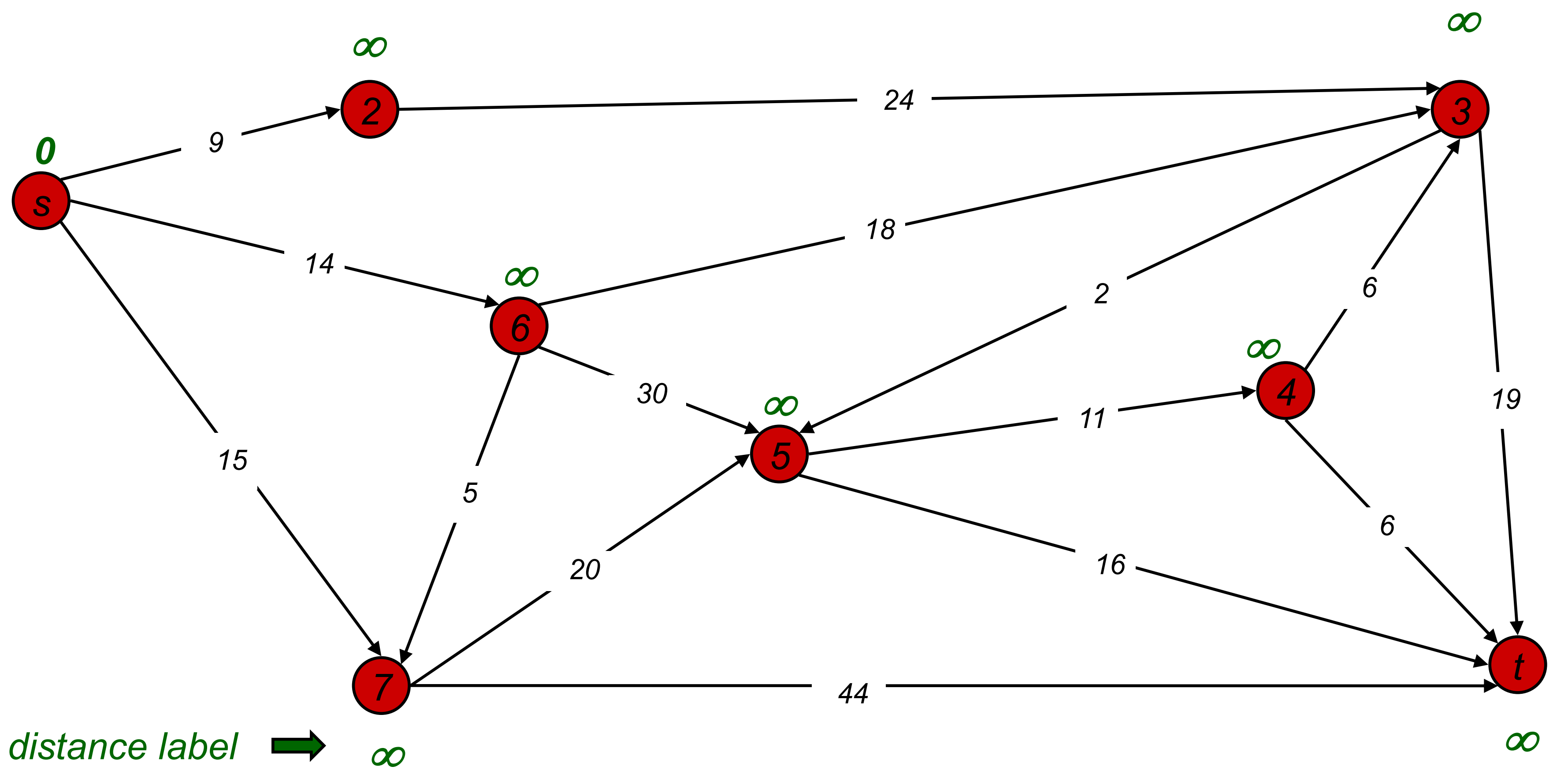
- Find shortest path from s to t.



# Dijkstra's Shortest Path Algorithm

*S* = { }

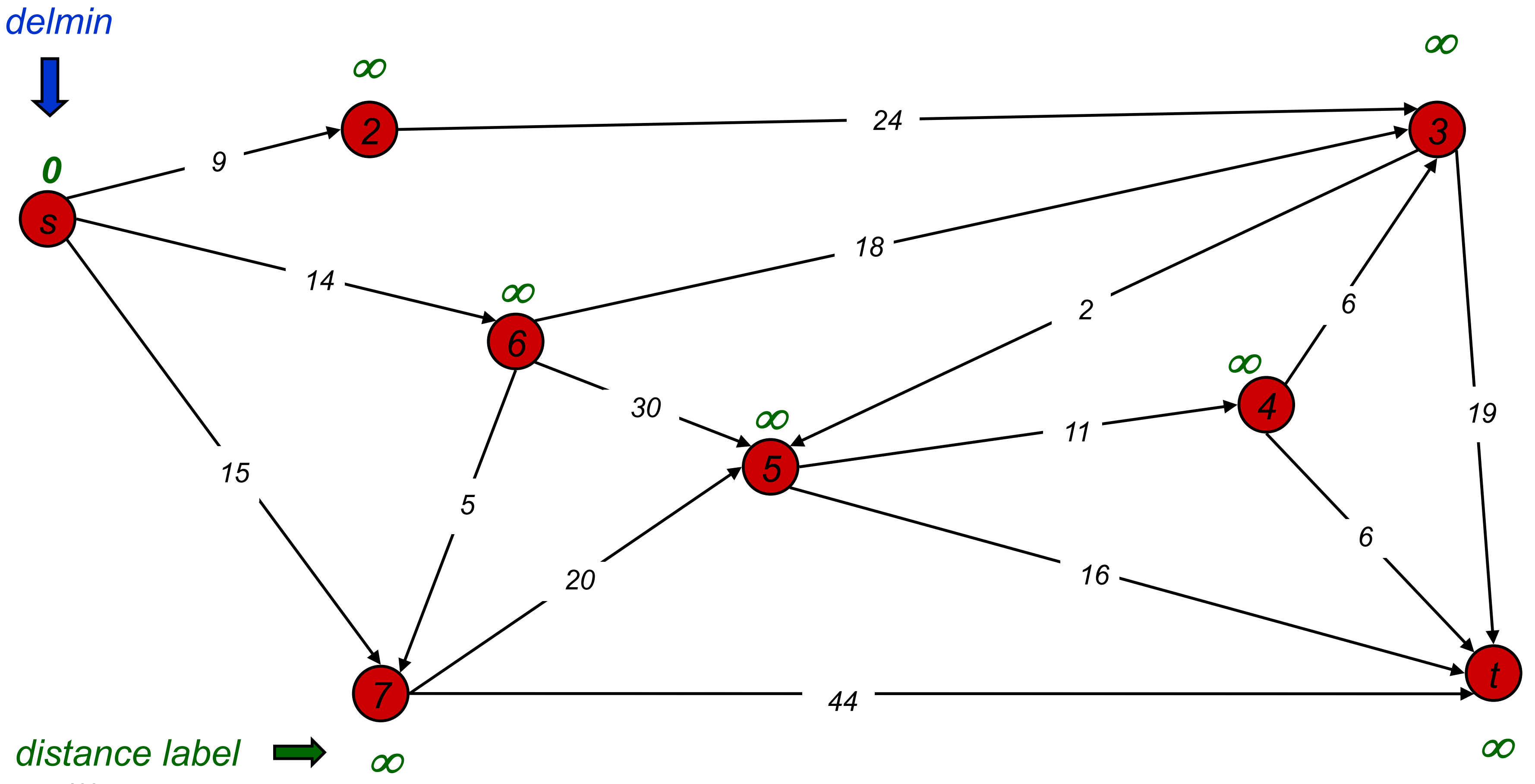
*PQ* = { *s, 2, 3, 4, 5, 6, 7, t* }



# Dijkstra's Shortest Path Algorithm

*S* = { }

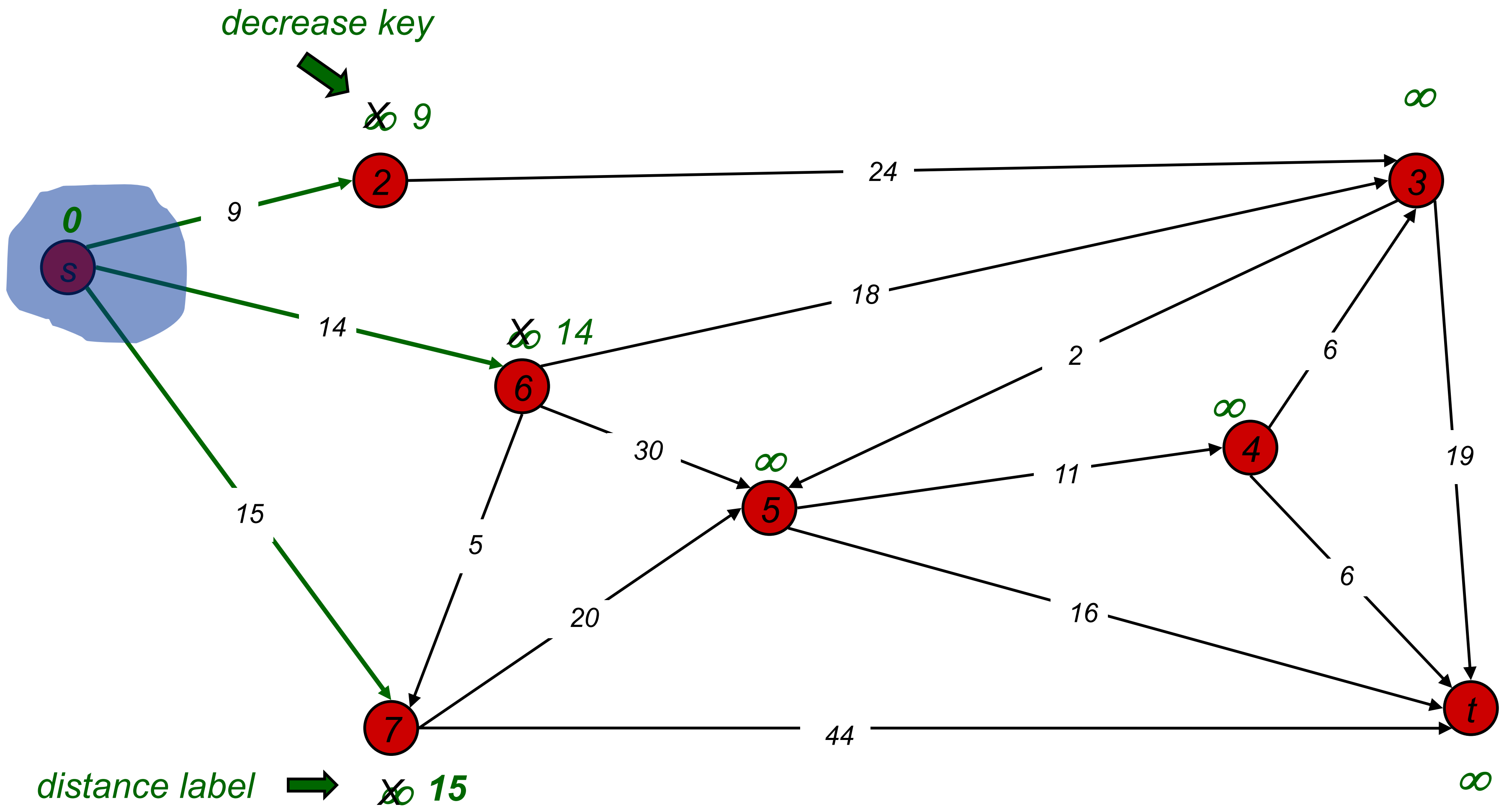
*PQ* = { *s*, 2, 3, 4, 5, 6, 7, *t* }



# Dijkstra's Shortest Path Algorithm

$S = \{ s \}$

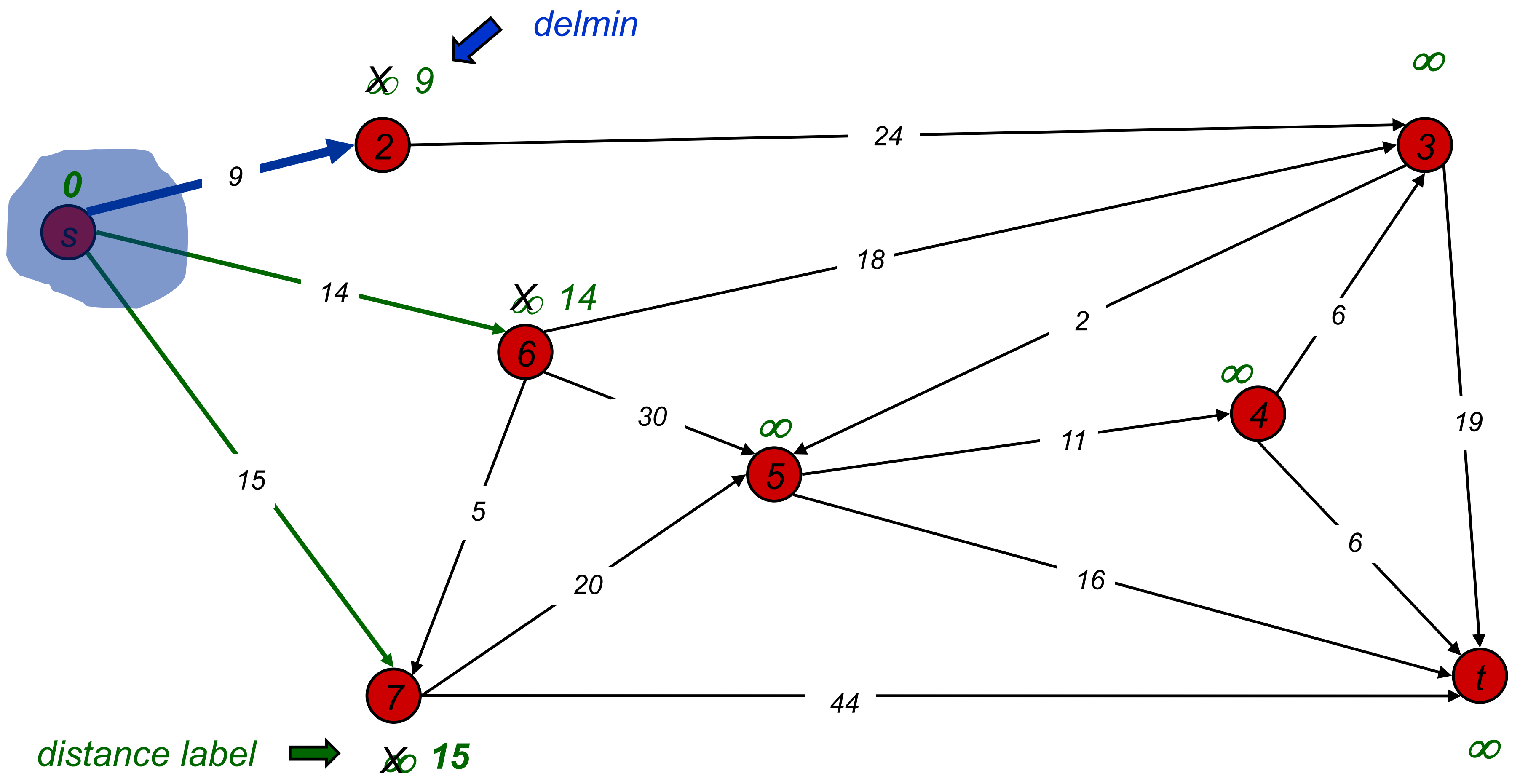
$PQ = \{ 2, 3, 4, 5, 6, 7, t \}$



# Dijkstra's Shortest Path Algorithm

$S = \{ s \}$

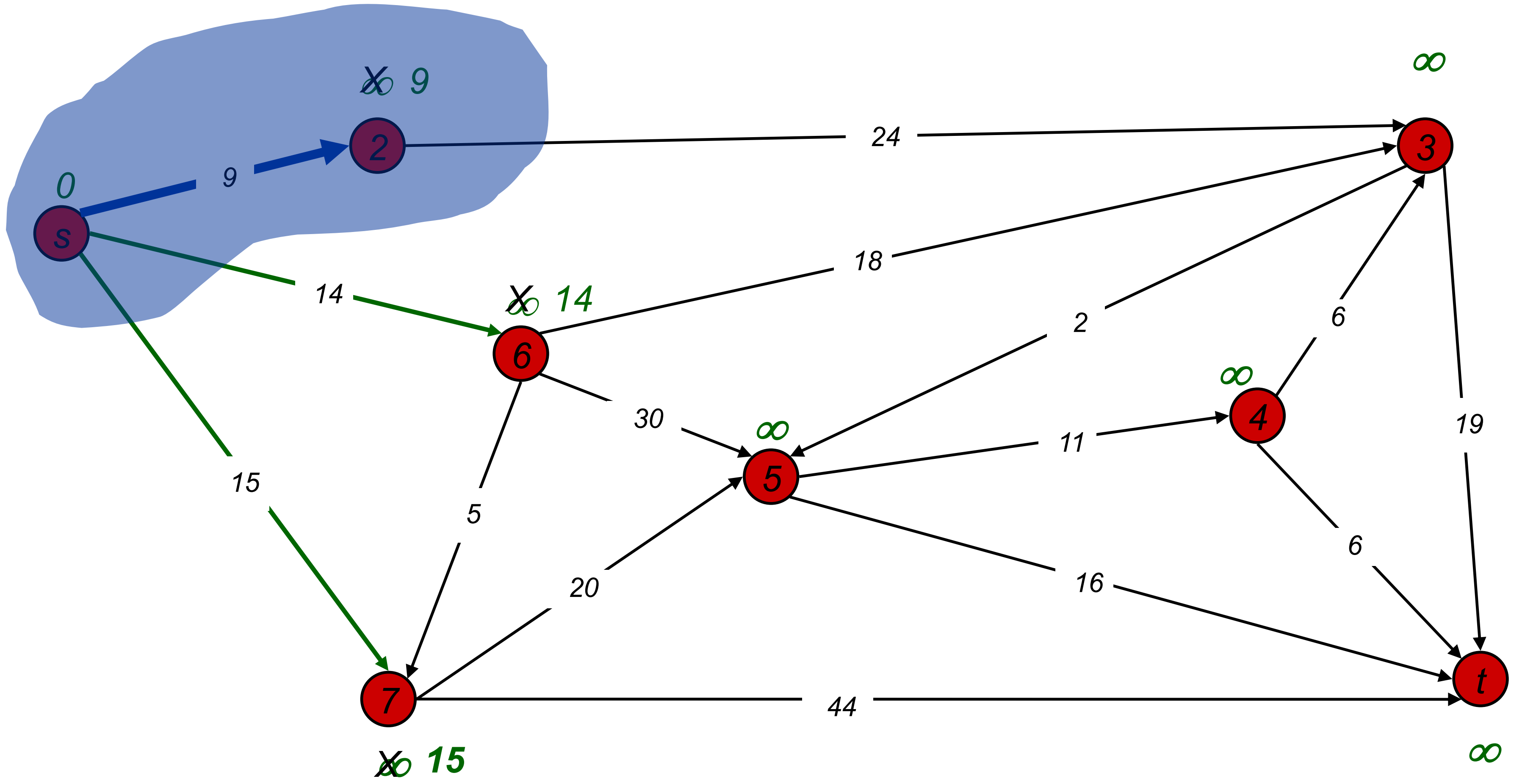
$PQ = \{ 2, 3, 4, 5, 6, 7, t \}$



# Dijkstra's Shortest Path Algorithm

*S* = { *s*, 2 }

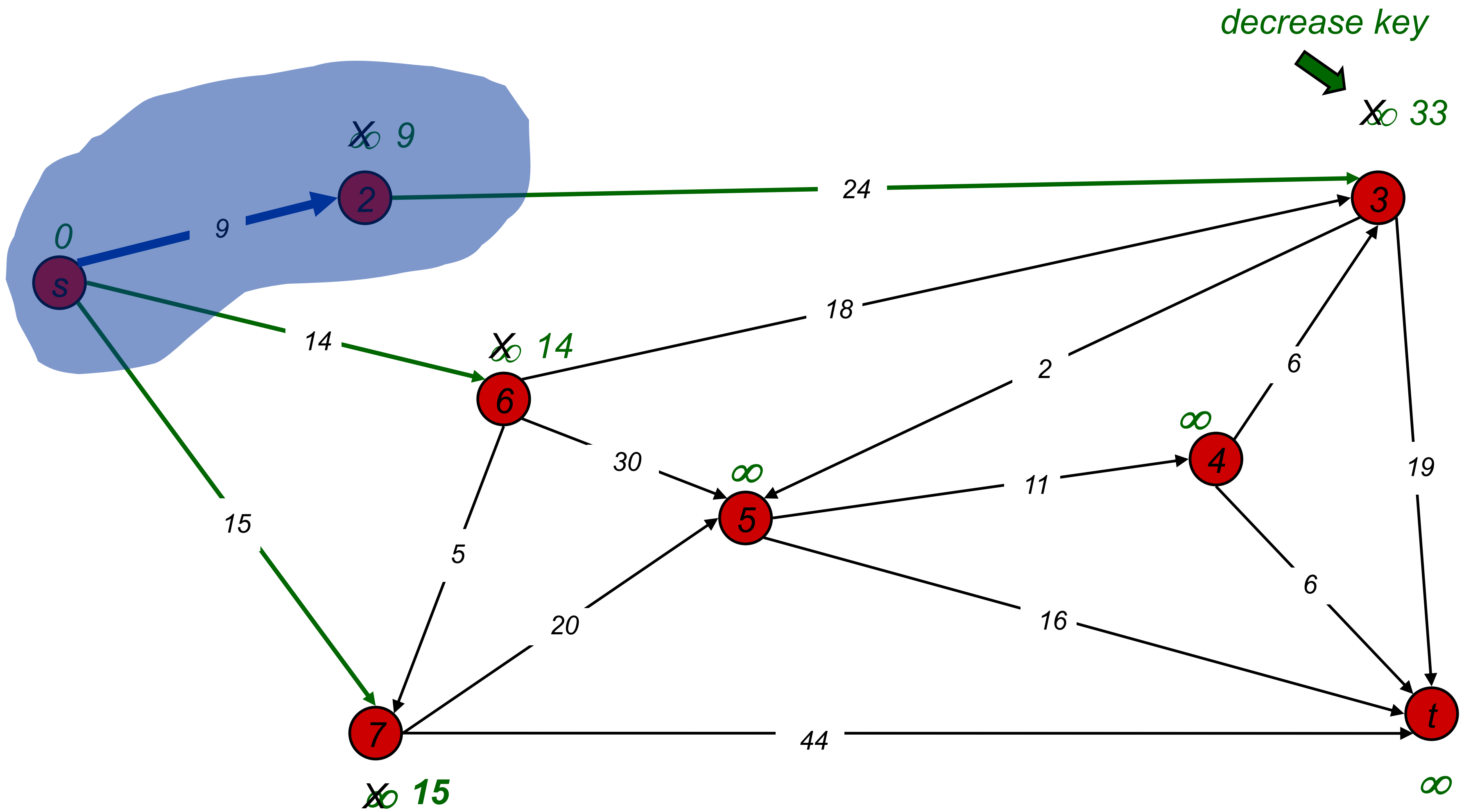
*PQ* = { 3, 4, 5, 6, 7, *t* }



# Dijkstra's Shortest Path Algorithm

*S* = { *s*, 2 }

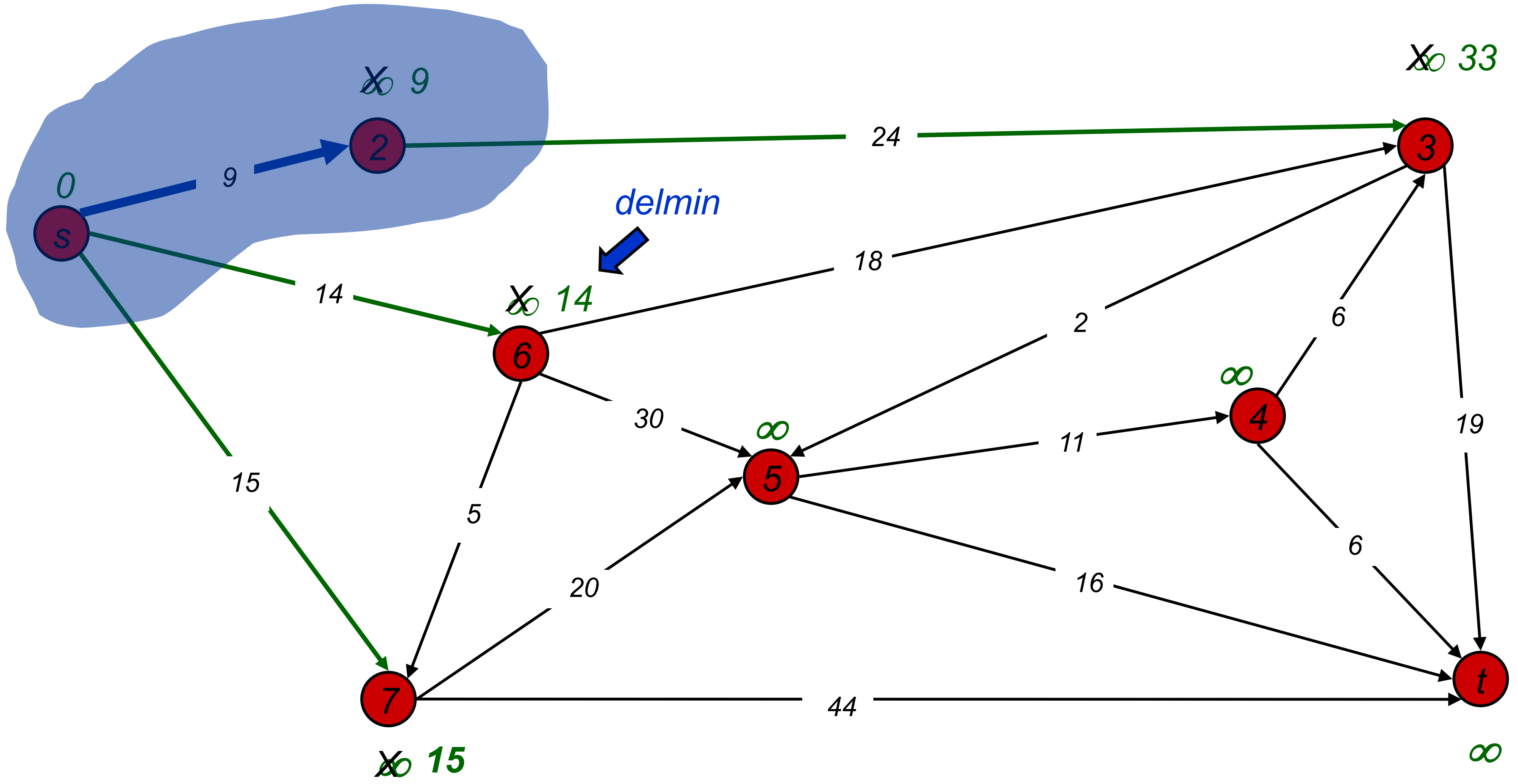
*PQ* = { 3, 4, 5, 6, 7, *t* }



# Dijkstra's Shortest Path Algorithm

*S* = { *s*, 2 }

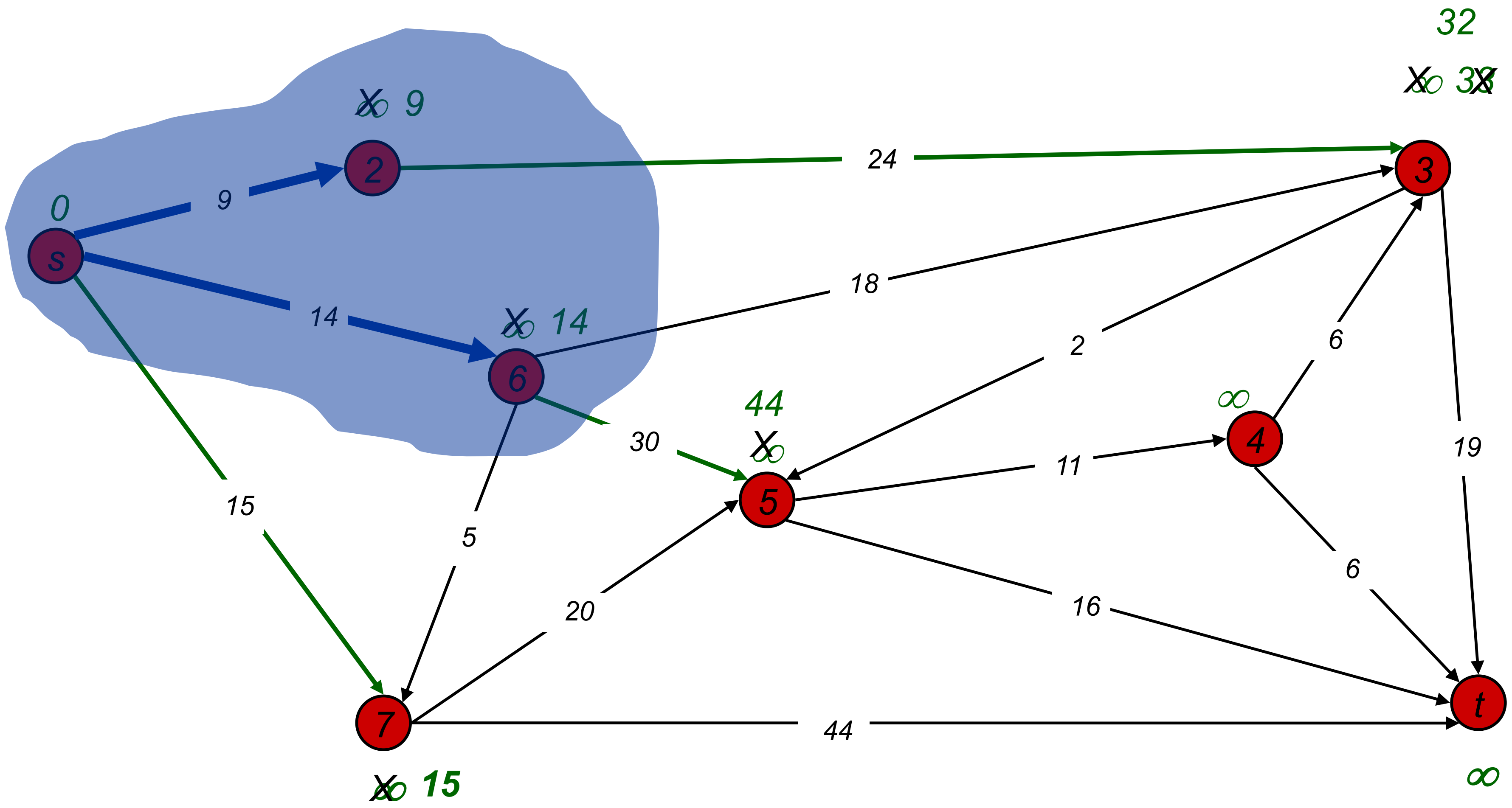
*PQ* = { 3, 4, 5, 6, 7, *t* }



# Dijkstra's Shortest Path Algorithm

*S* = { *s*, 2, 6 }

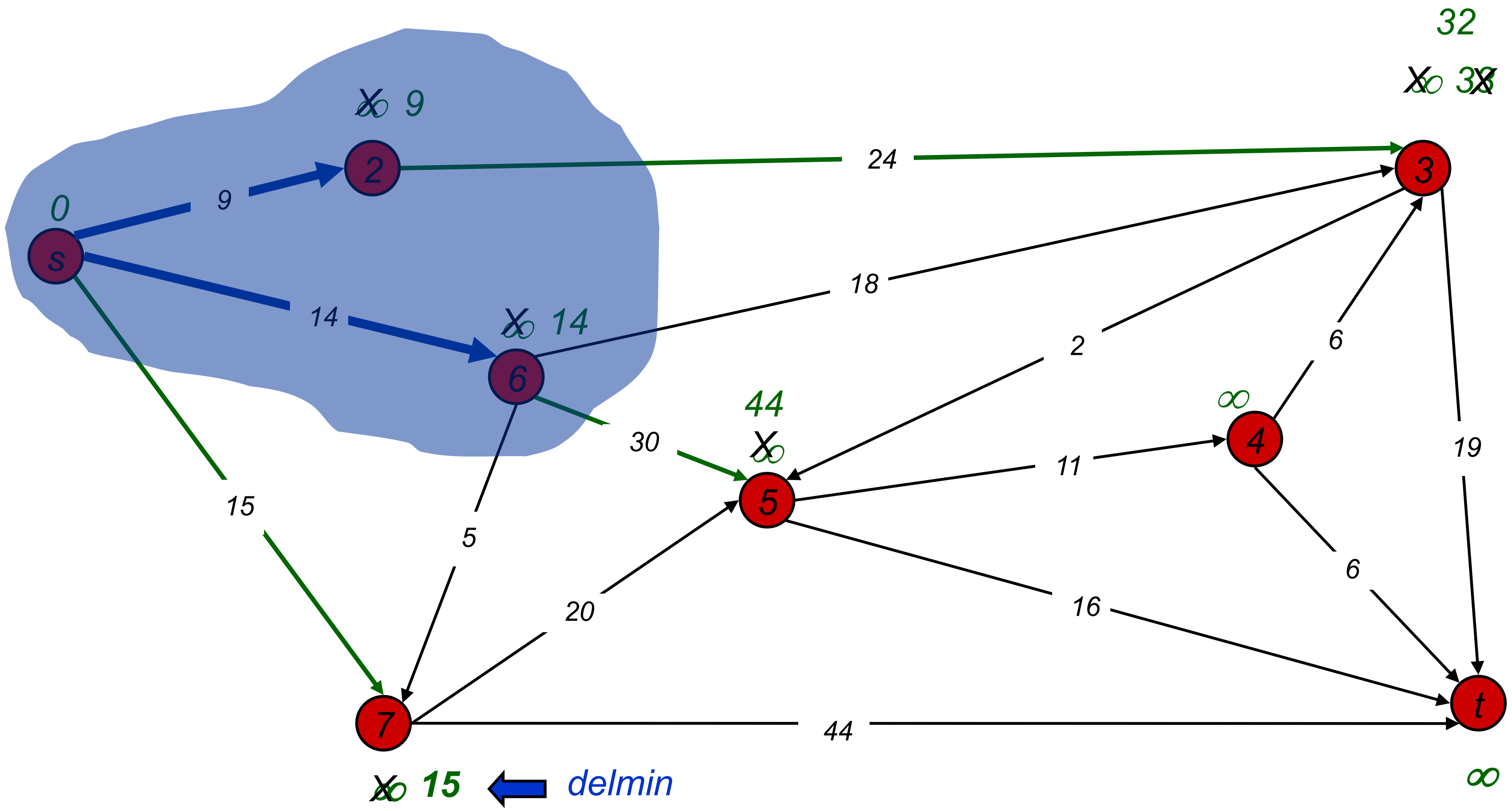
*PQ* = { 3, 4, 5, 7, *t* }



# Dijkstra's Shortest Path Algorithm

*S* = { *s*, 2, 6 }

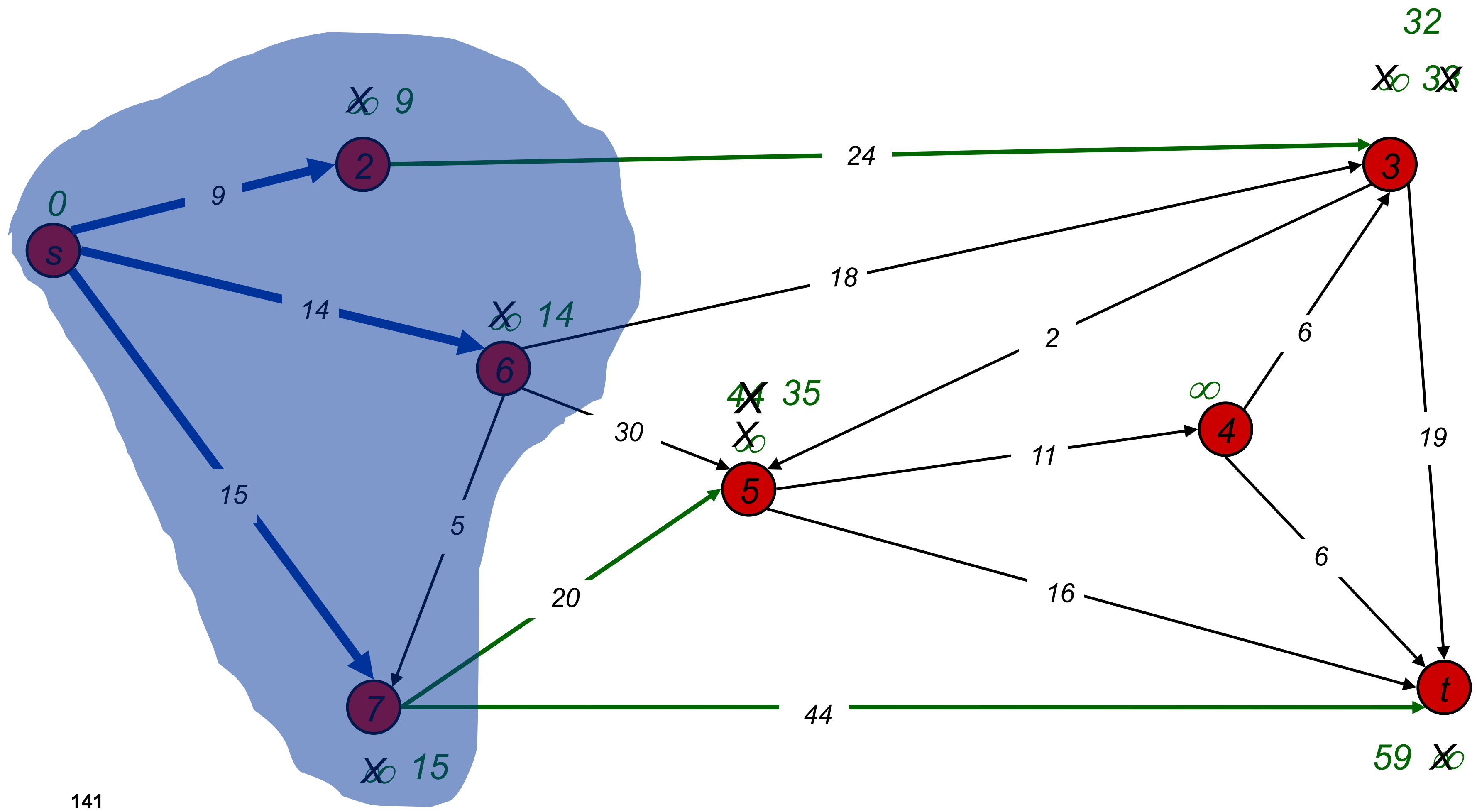
*PQ* = { 3, 4, 5, 7, *t* }



# Dijkstra's Shortest Path Algorithm

*S* = { *s*, 2, 6, 7 }

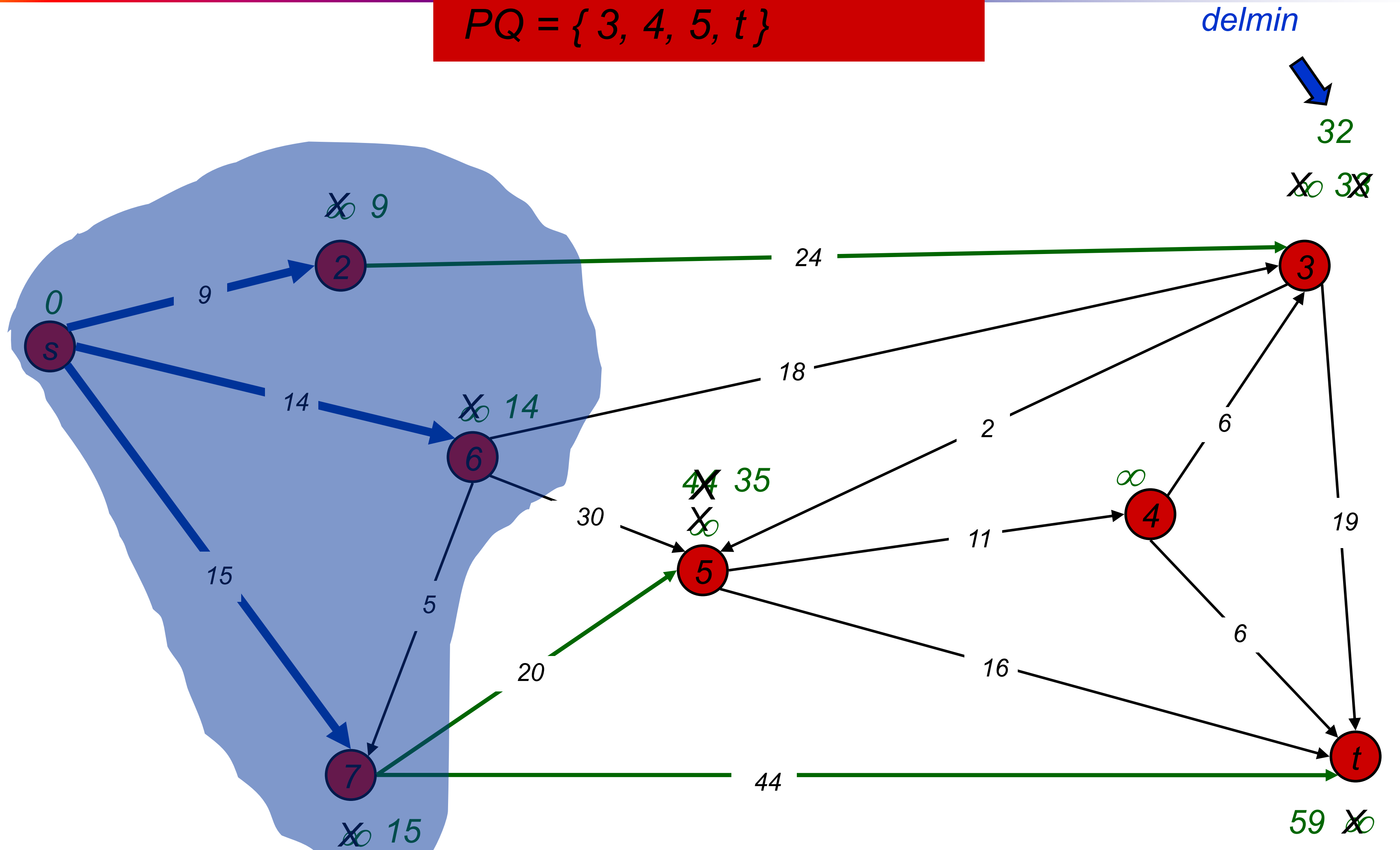
*PQ* = { 3, 4, 5, *t* }



# Dijkstra's Shortest Path Algorithm

*S* = { *s*, 2, 6, 7 }

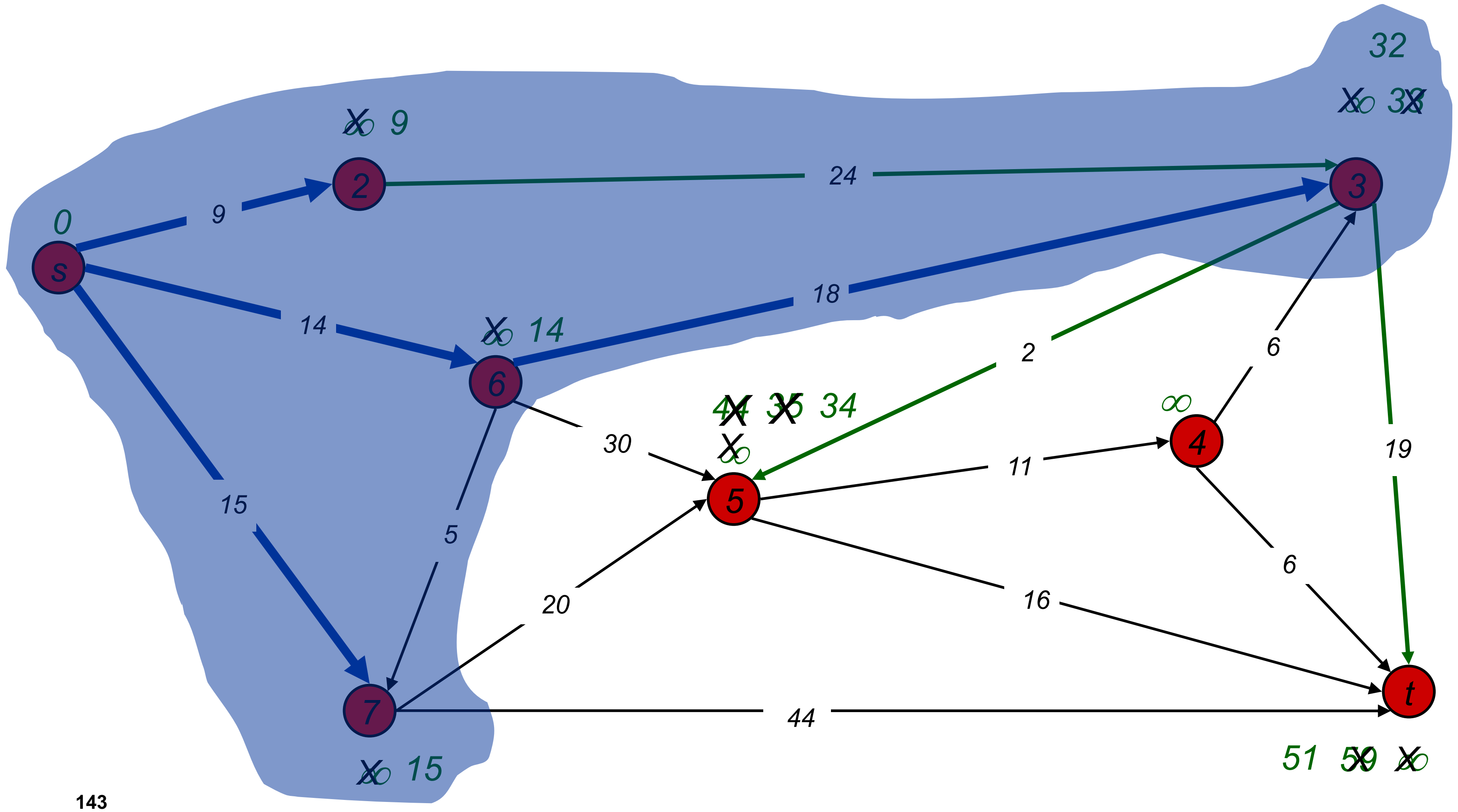
*PQ* = { 3, 4, 5, *t* }



# Dijkstra's Shortest Path Algorithm

$S = \{ s, 2, 3, 6, 7 \}$

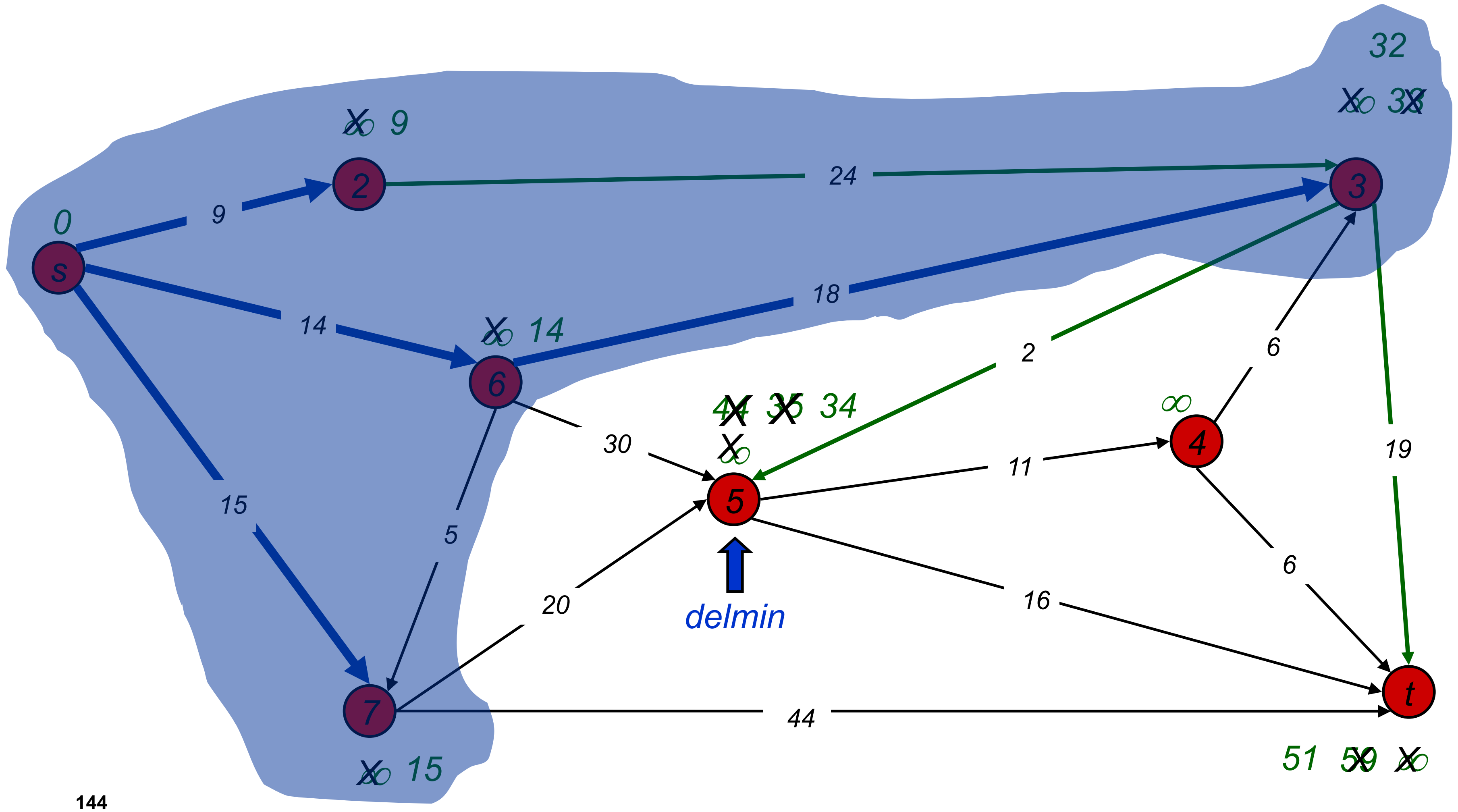
$PQ = \{ 4, 5, t \}$



# Dijkstra's Shortest Path Algorithm

S = { s, 2, 3, 6, 7 }

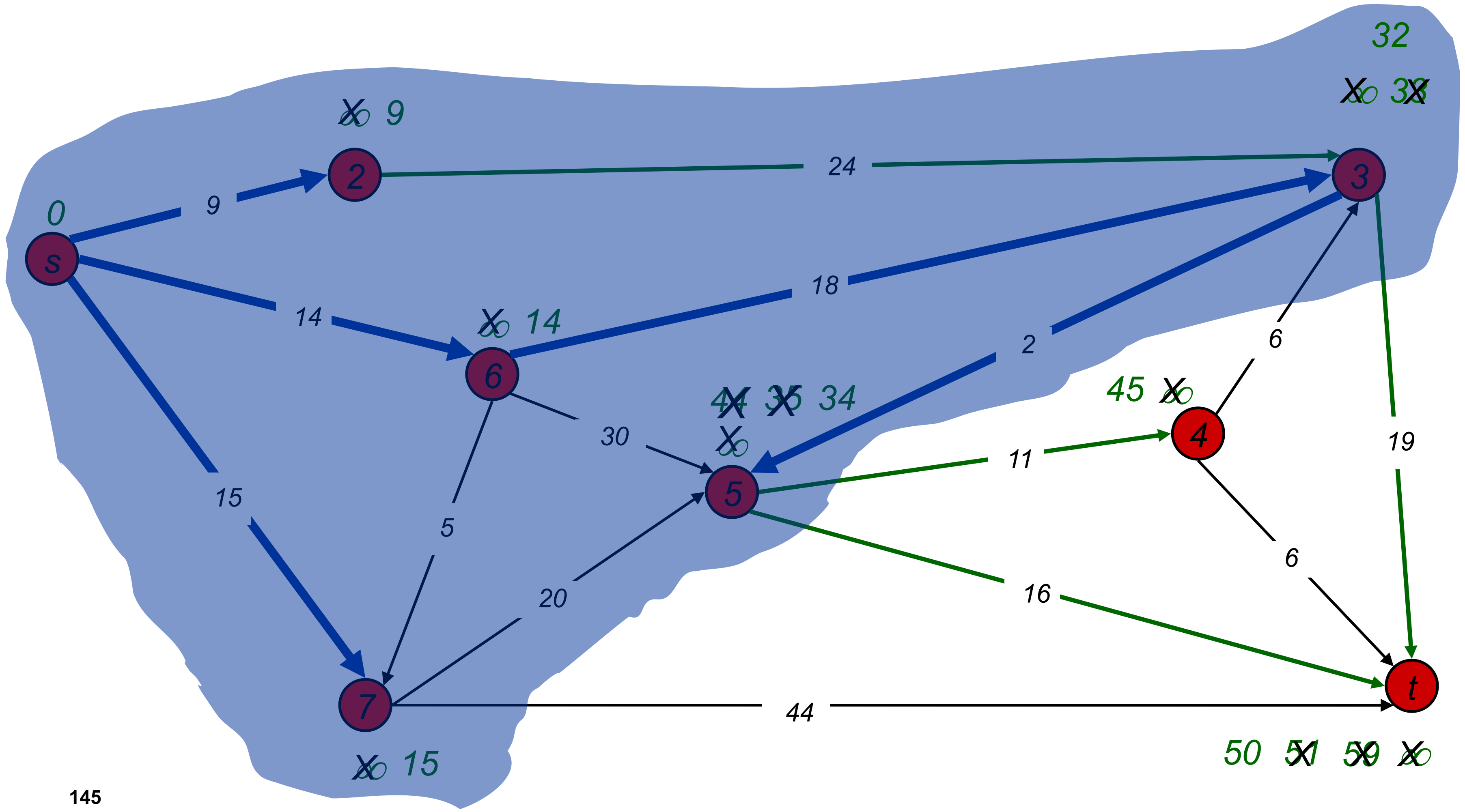
PQ = { 4, 5, t }



# Dijkstra's Shortest Path Algorithm

*S* = { *s*, 2, 3, 5, 6, 7 }

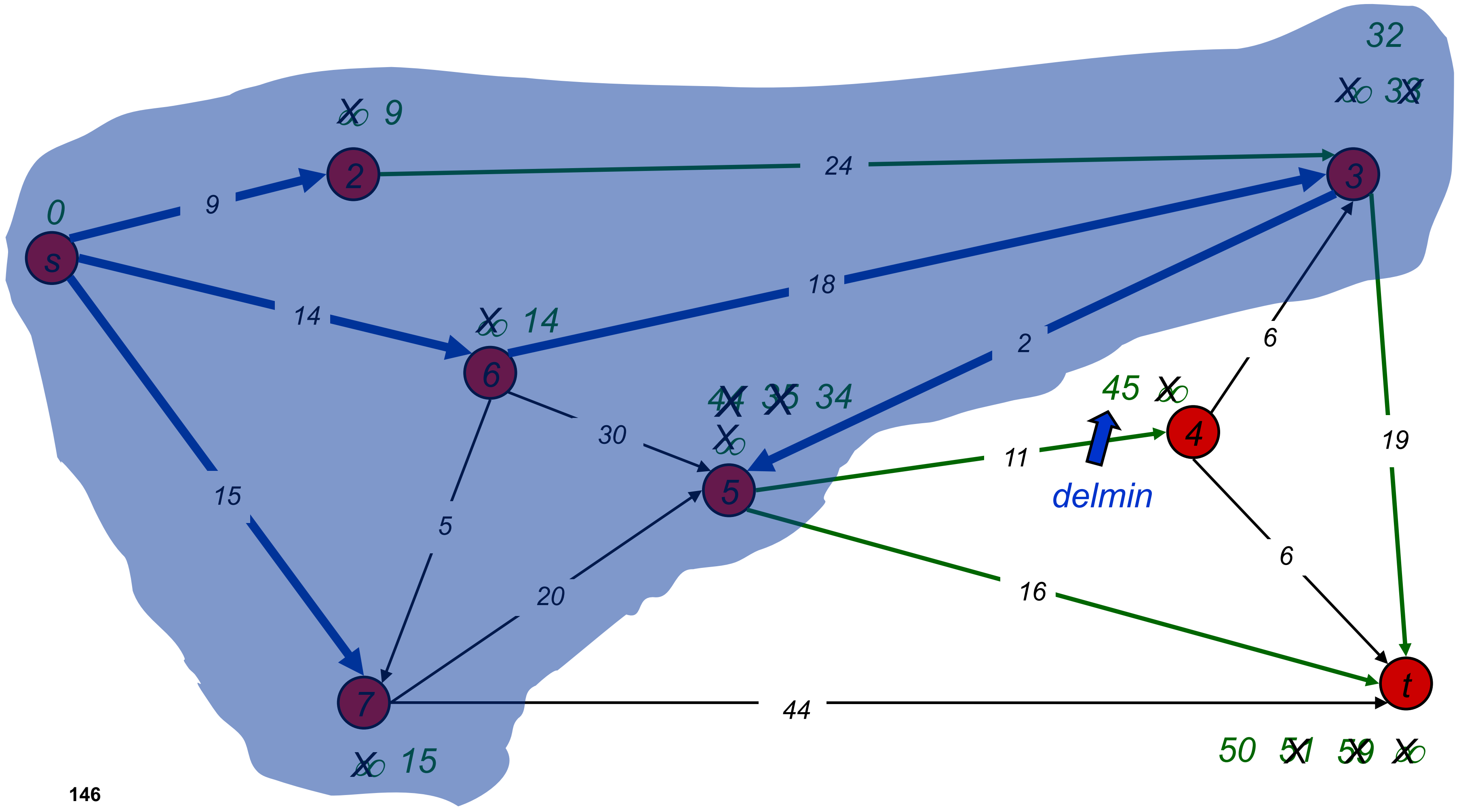
*PQ* = { 4, *t* }



# Dijkstra's Shortest Path Algorithm

*S* = { *s*, 2, 3, 5, 6, 7 }

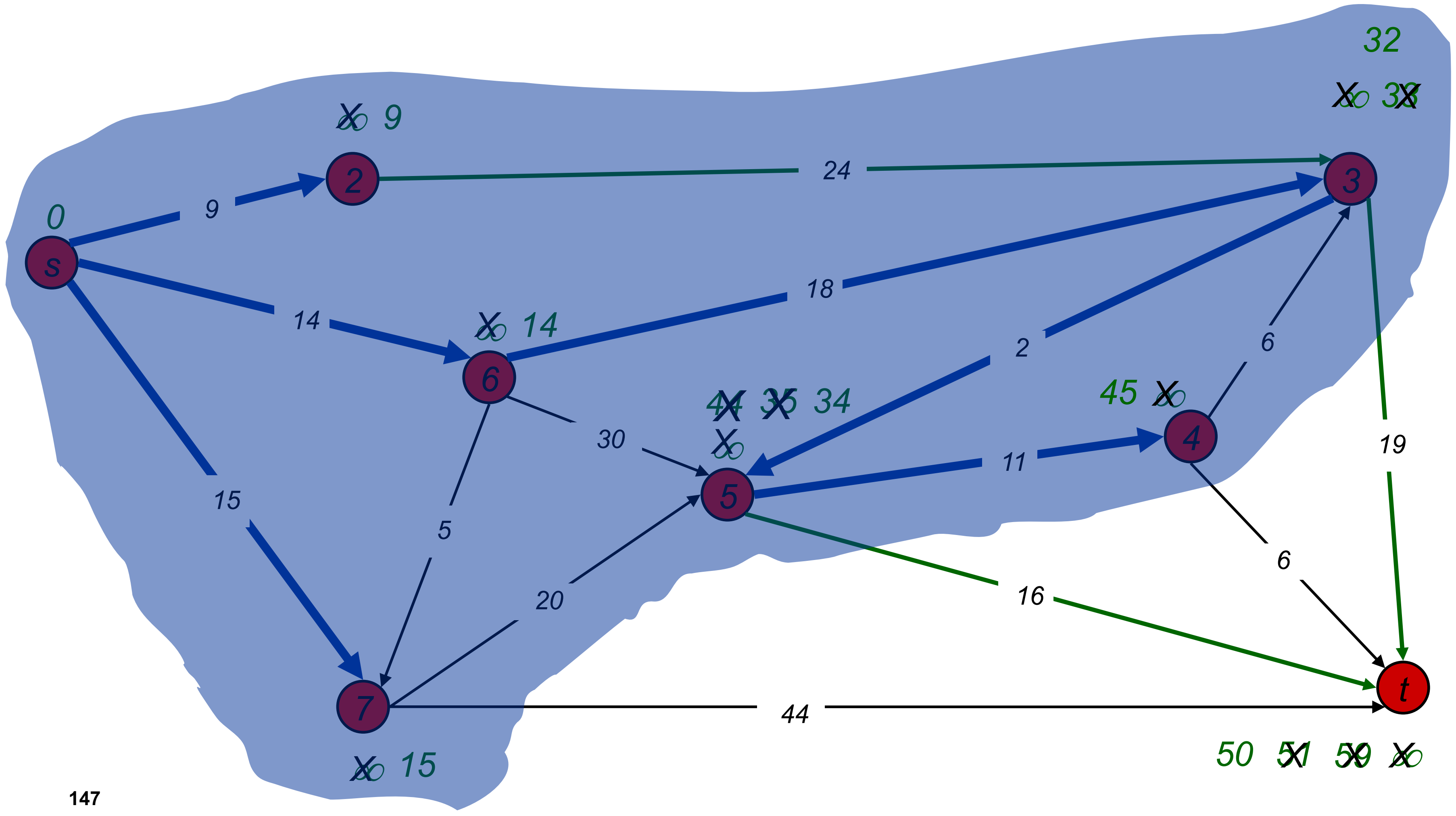
*PQ* = { 4, *t* }



# Dijkstra's Shortest Path Algorithm

$S = \{ s, 2, 3, 4, 5, 6, 7 \}$

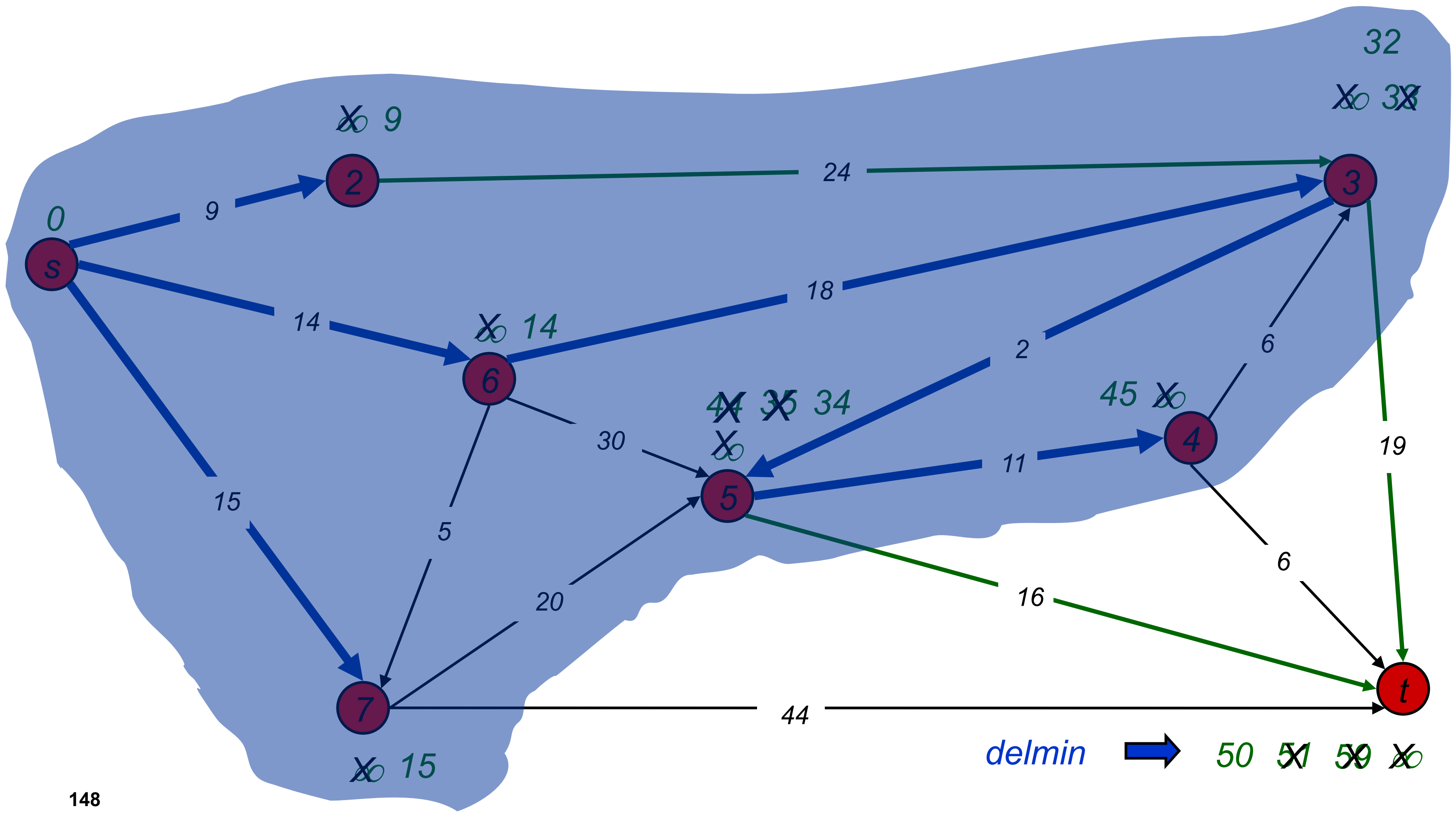
$PQ = \{ t \}$



# Dijkstra's Shortest Path Algorithm

*S* = { *s*, 2, 3, 4, 5, 6, 7 }

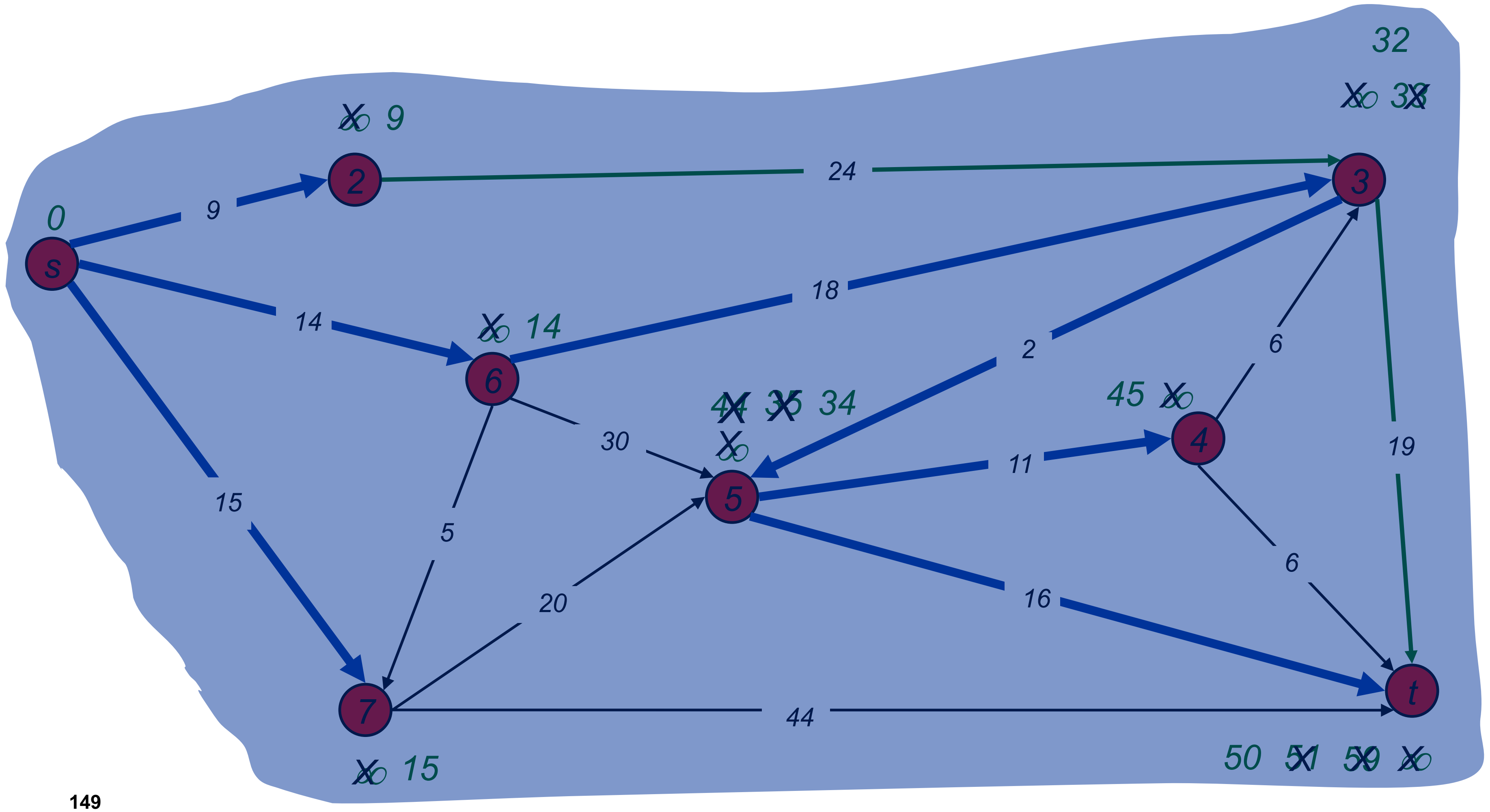
*PQ* = { *t* }



# Dijkstra's Shortest Path Algorithm

*S* = { *s*, 2, 3, 4, 5, 6, 7, *t* }

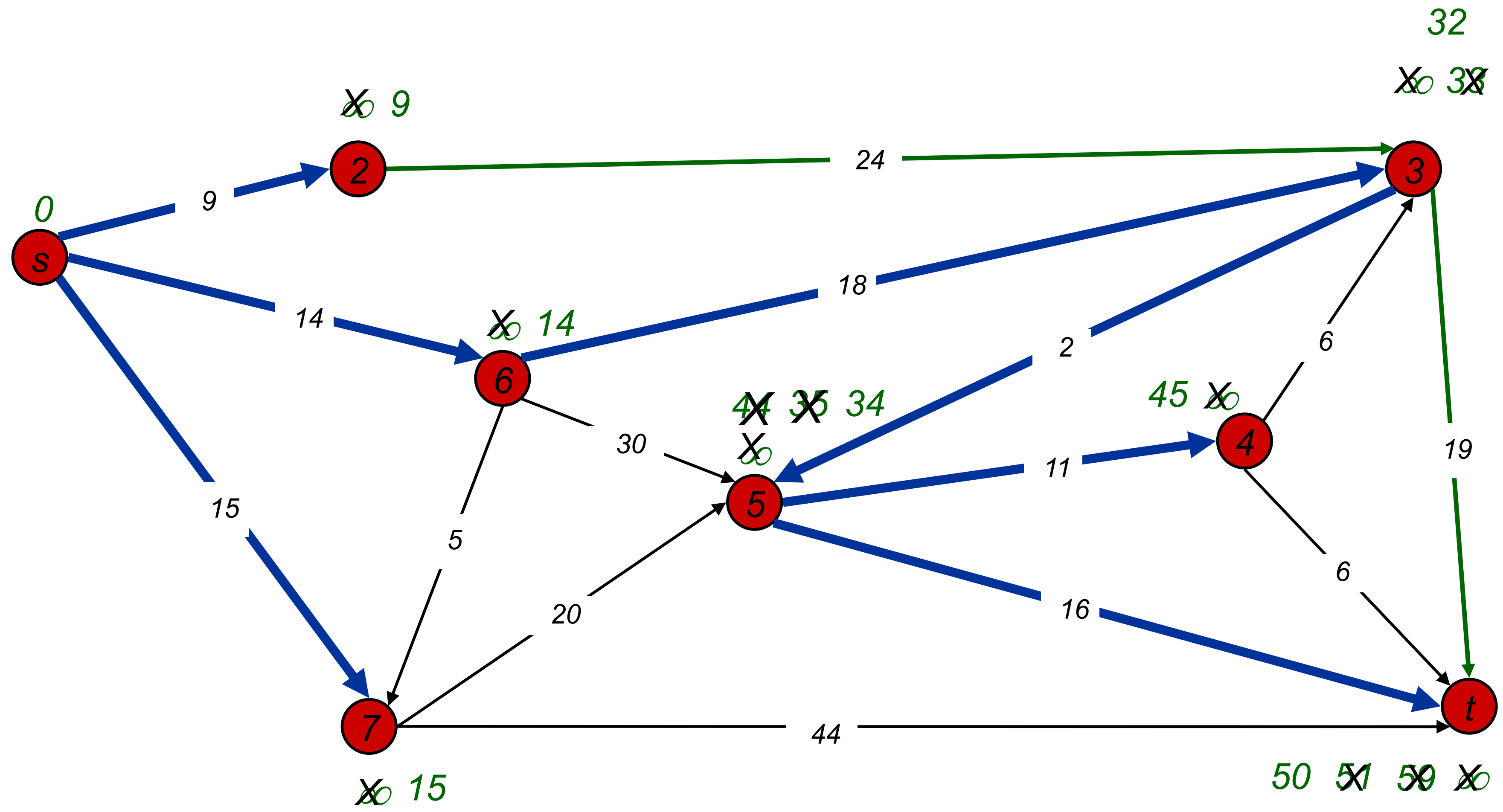
*PQ* = { }



# Dijkstra's Shortest Path Algorithm

*S* = { *s*, *2*, *3*, *4*, *5*, *6*, *7*, *t* }

*PQ* = { }

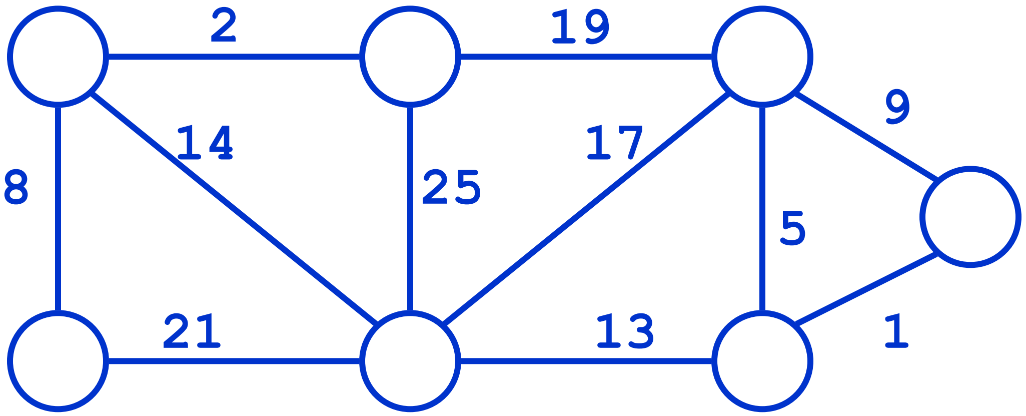


## Kruskal's Algorithm

```
Kruskal()
{
    T = $\emptyset$;
    for each v $\in$ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) $\in$ E (in sorted order)
        if FindSet(u) $\neq$ FindSet(v)
            T = T $\cup$ {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

# Kruskal's Algorithm

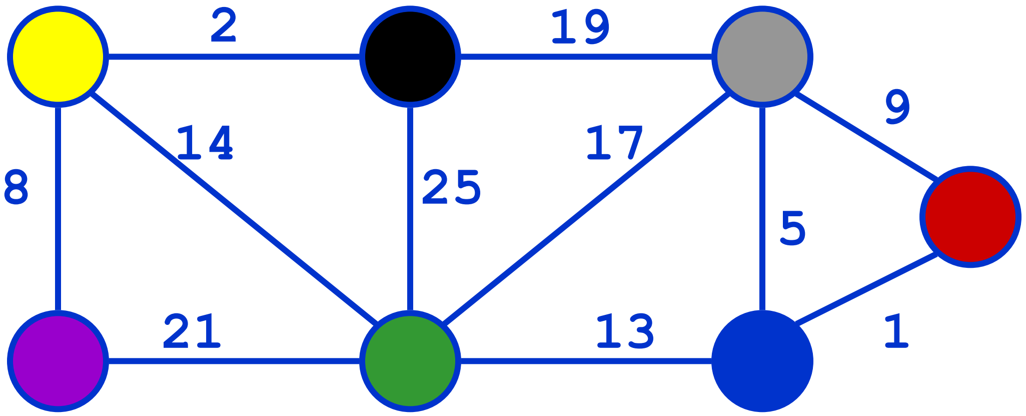
*Run the algorithm:*



```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

# Kruskal's Algorithm

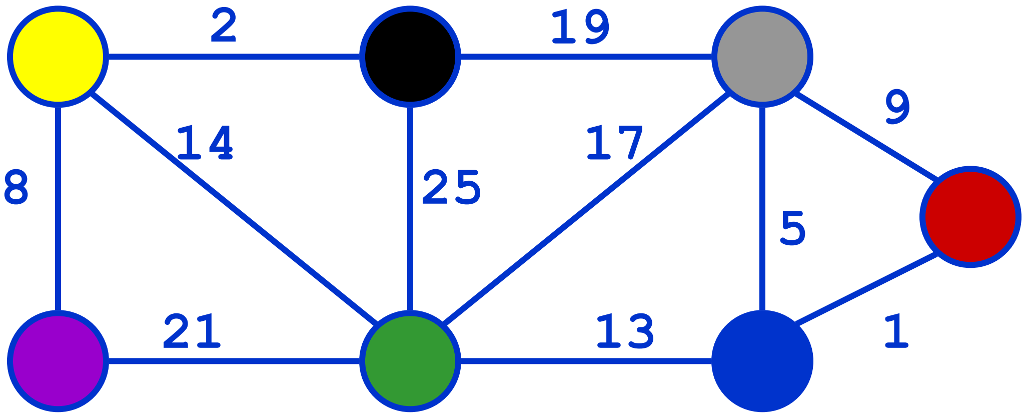
*Run the algorithm:*



```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

# Kruskal's Algorithm

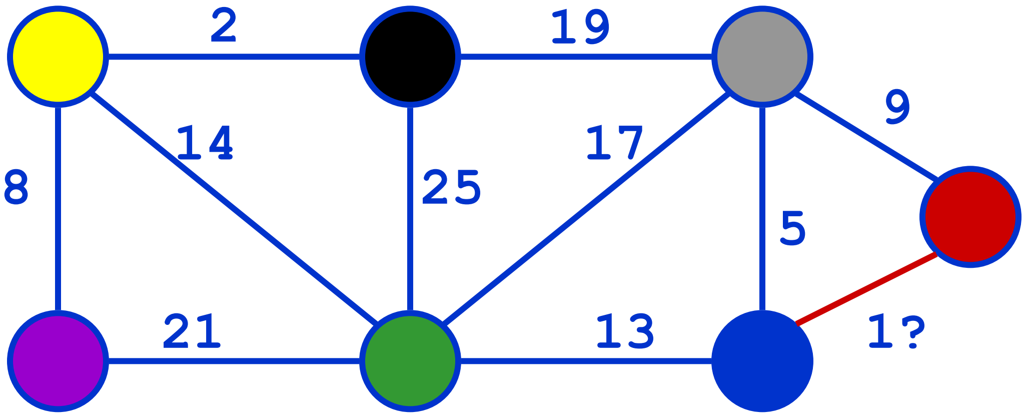
*Run the algorithm:*



```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
  { sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

# Kruskal's Algorithm

*Run the algorithm:*

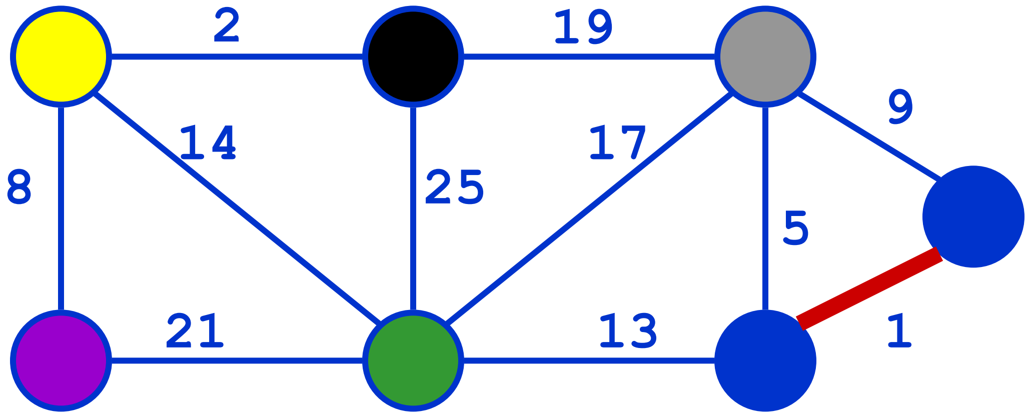


```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

# Kruskal's Algorithm

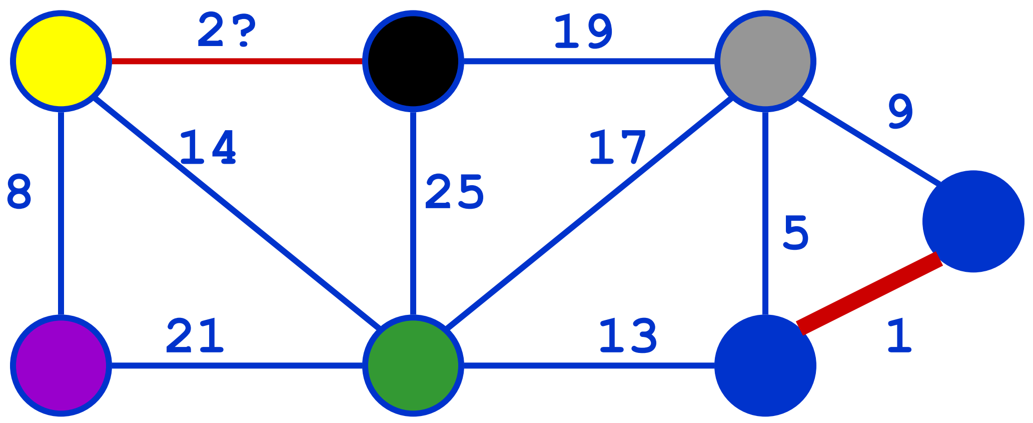
```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

***Run the algorithm:***



# Kruskal's Algorithm

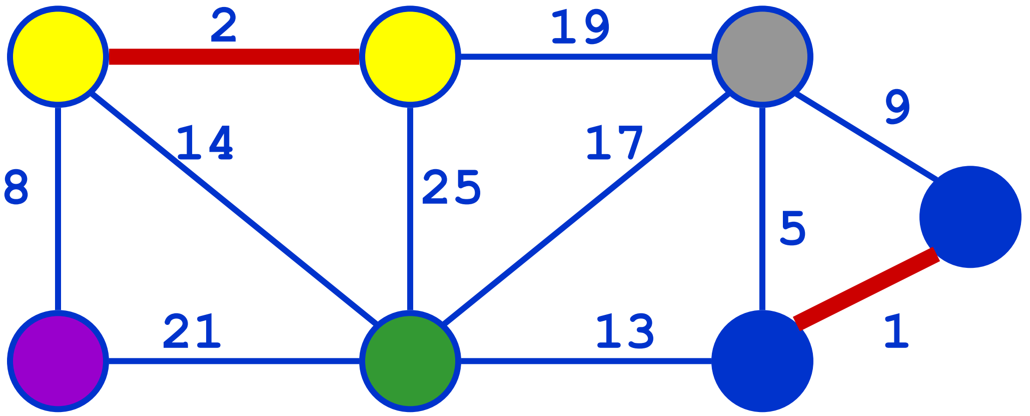
*Run the algorithm:*



```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

# Kruskal's Algorithm

*Run the algorithm:*

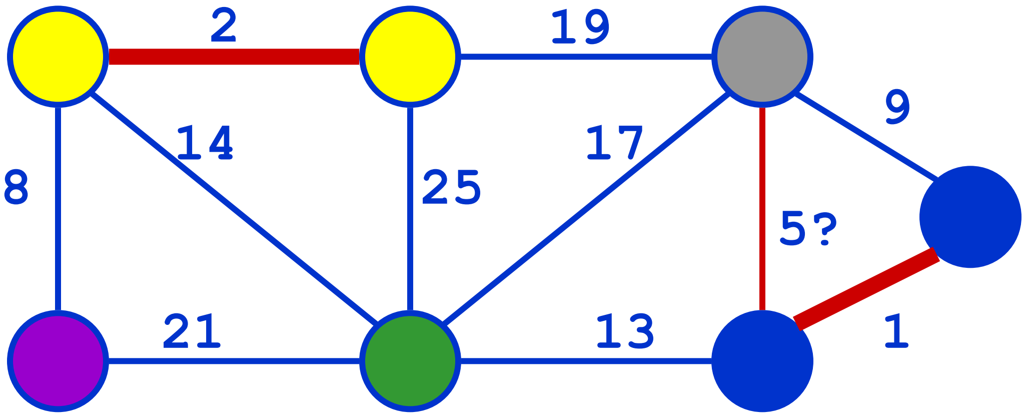


```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

# Kruskal's Algorithm

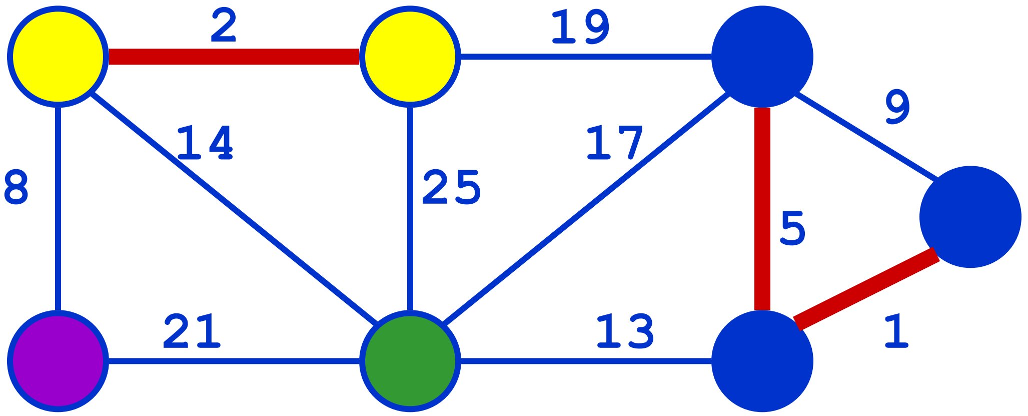
```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

*Run the algorithm:*



# Kruskal's Algorithm

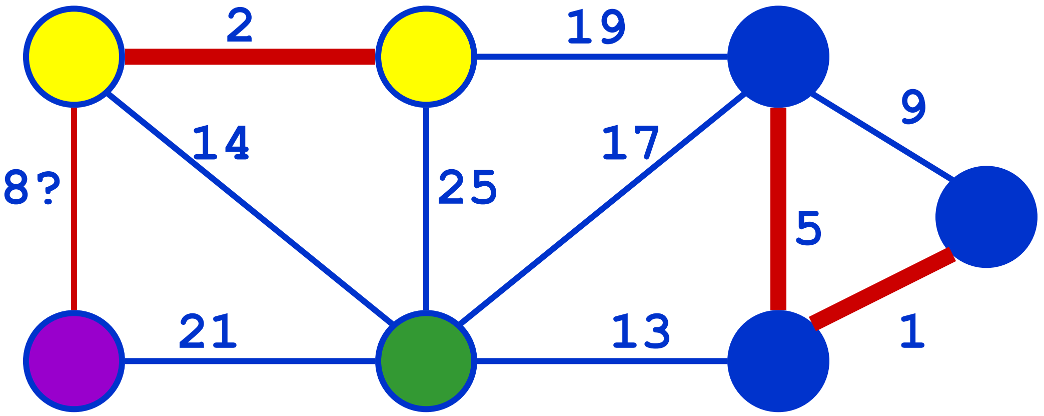
***Run the algorithm:***



```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

# Kruskal's Algorithm

*Run the algorithm:*

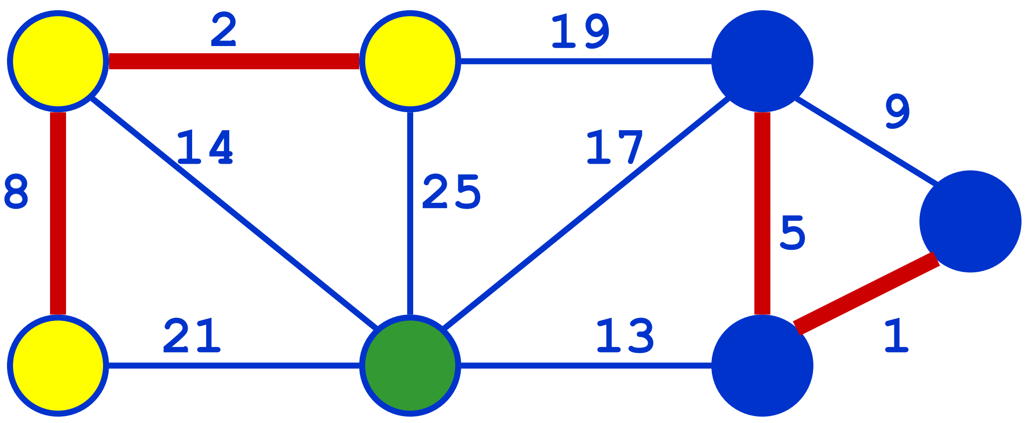


```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T ∪ {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

# Kruskal's Algorithm

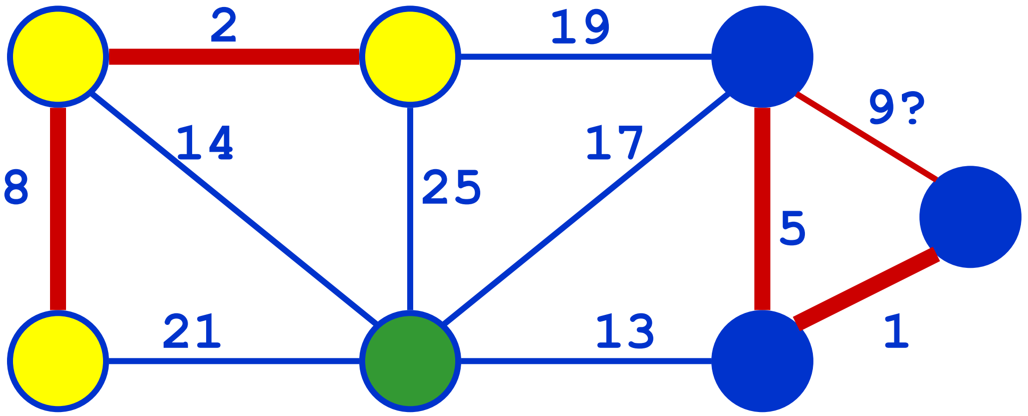
```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

*Run the algorithm:*



# Kruskal's Algorithm

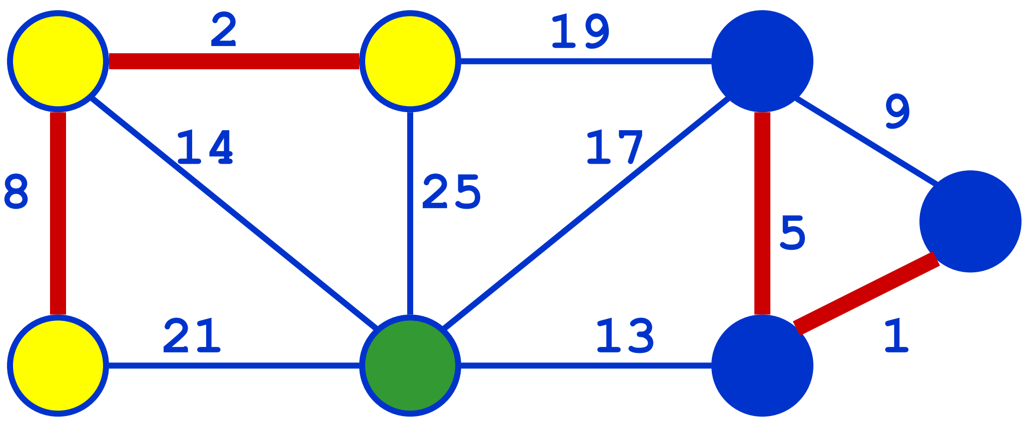
***Run the algorithm:***



```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

# Kruskal's Algorithm

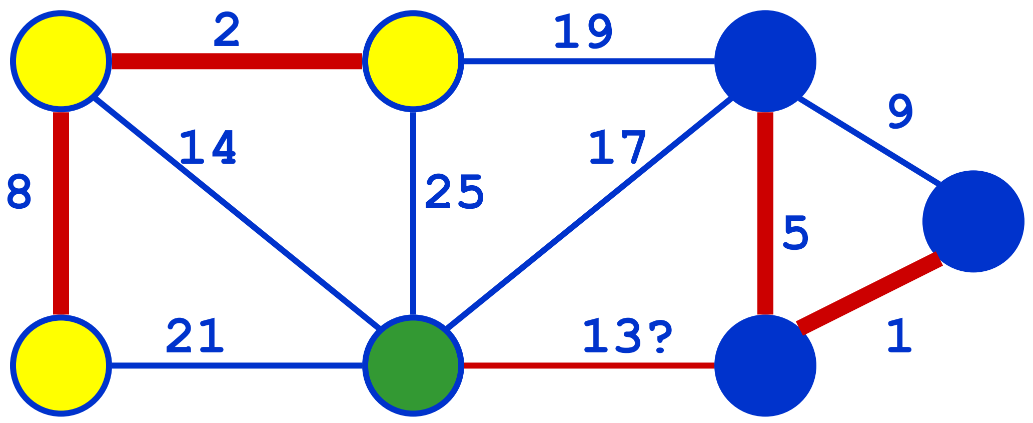
*Run the algorithm:*



```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

# Kruskal's Algorithm

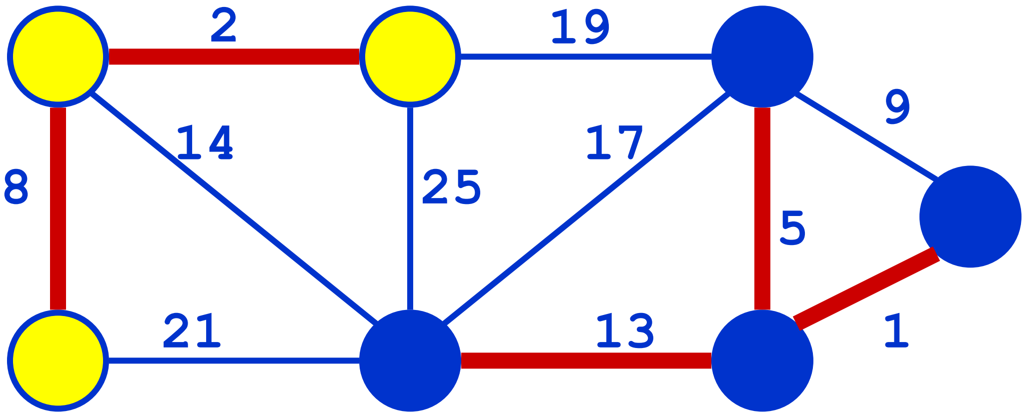
***Run the algorithm:***



```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

# Kruskal’s Algorithm

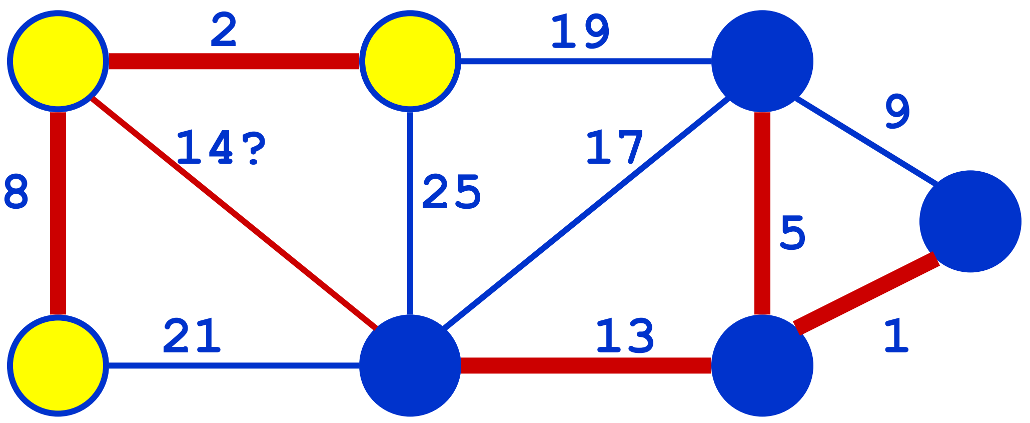
***Run the algorithm:***



```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

# Kruskal's Algorithm

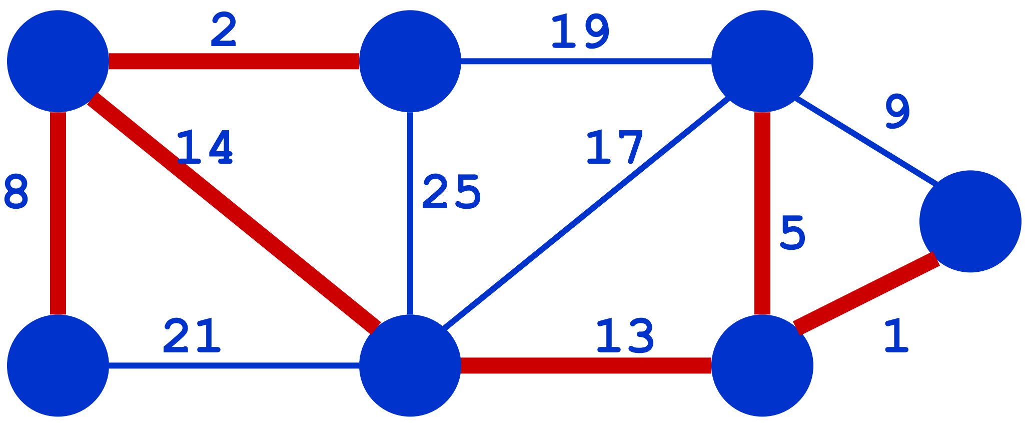
*Run the algorithm:*



```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

# Kruskal’s Algorithm

***Run the algorithm:***

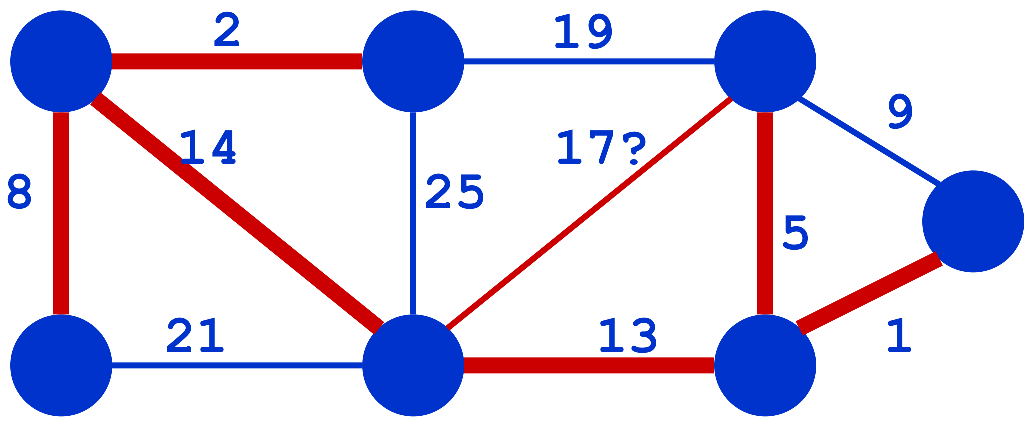


```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T ∪ {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

# Kruskal's Algorithm

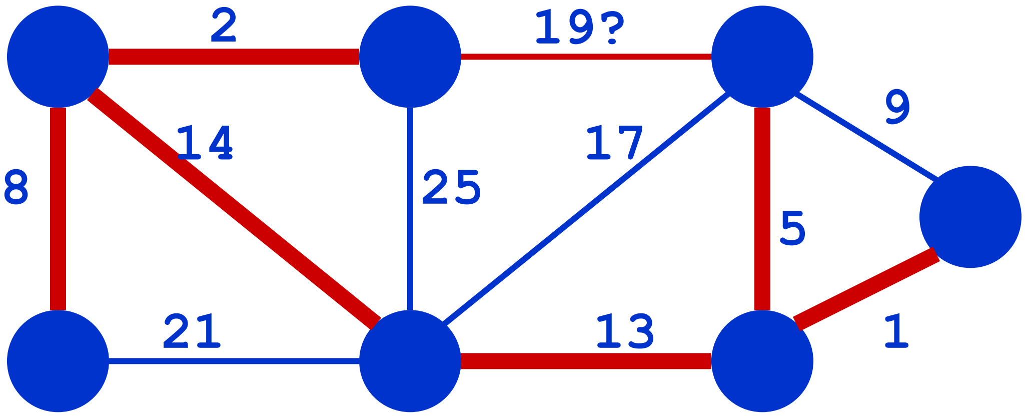
```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

***Run the algorithm:***



# Kruskal's Algorithm

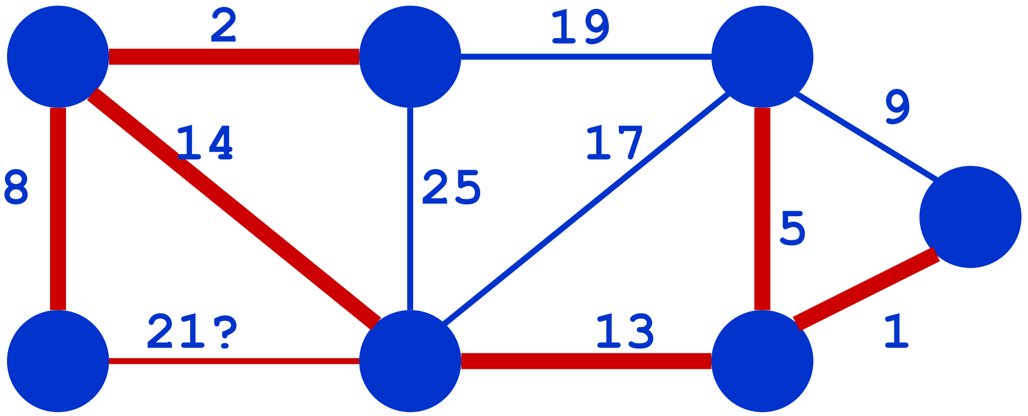
***Run the algorithm:***



```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

# Kruskal's Algorithm

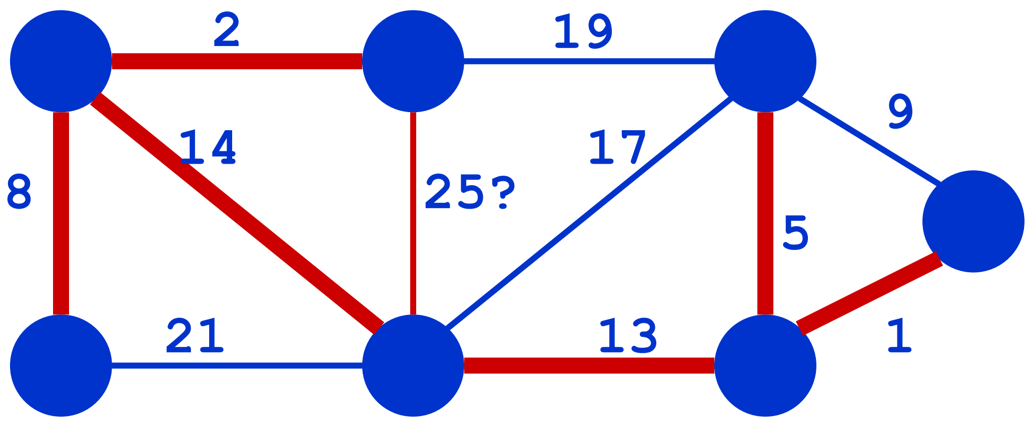
***Run the algorithm:***



```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

# Kruskal’s Algorithm

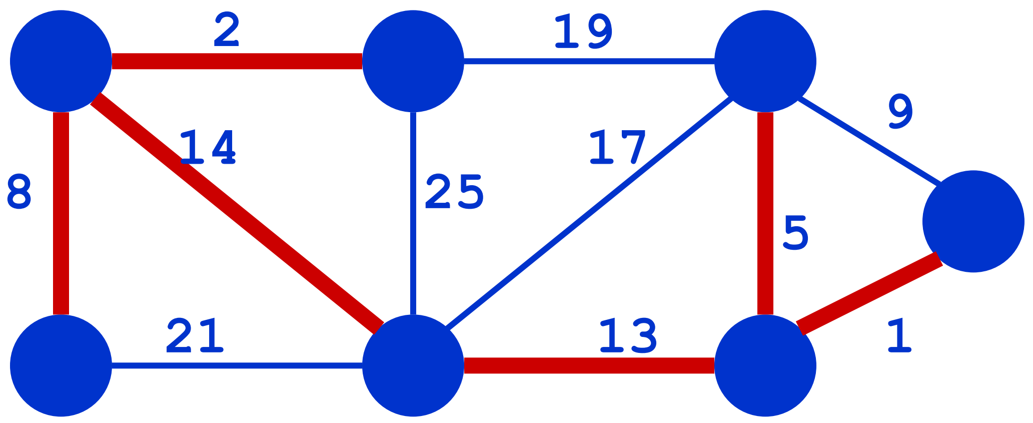
*Run the algorithm:*



```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

# Kruskal's Algorithm

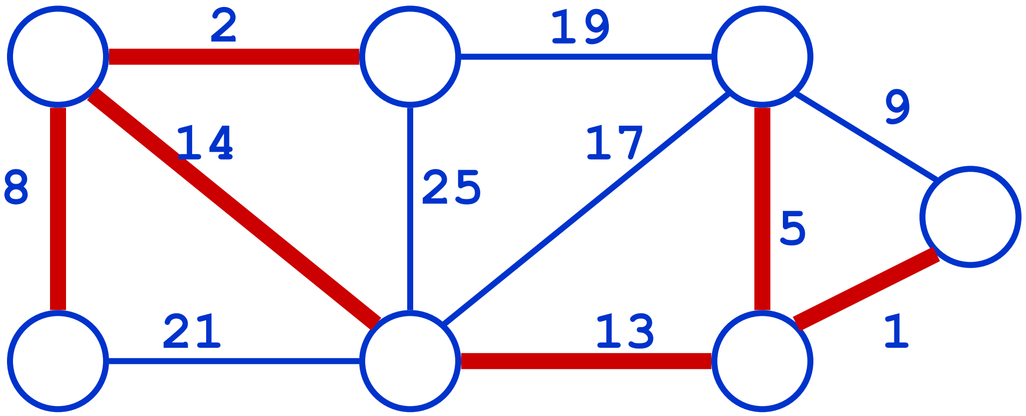
***Run the algorithm:***



```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

# Kruskal’s Algorithm

*Run the algorithm:*



```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```